Tornsdorf

AMIGA

Maschinen Sprache für Einsteiger

DATA BECKER

Manfred Tornsdorf

Amiga Maschinensprache für Einsteiger

DATA BECKER

Copyright © 1990 by DATA BECKER GmbH
Merowingerstr. 30
4000 Düsseldorf 1

2. unveränderte Auflage 1990

Umschlaggestaltung Werner Leinhos

Textverarbeitung und Gestaltung Udo Bretschneider

Text verarbeitet mit Word 5.0, Microsoft

Druck und buchbinderische Verarbeitung Graf und Pflügge, Düsseldorf

ISBN 3-89011-172-6

Wichtiger Hinweis

Die in diesem Buch wiedergegebenen Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle technischen Angaben und Programme in diesem Buch wurden vom Autor mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Inhaltsverzeichnis

1. Ein paar notwendige Grundbegriffe 11

1.1 Von Bits, Bytes, Worten und Langworten... 11
1.2 Die unterschiedlichen Zahlensysteme 12
1.3 Der Speicher (RAM, ROM, WOM) 19
1.4 Die CPU - Das Wichtigste Ihres Computers 23

2. Es geht los - ein erstes Programm 29

2.1 Wir addieren Zahlen 29
2.2 Wir assemblieren das Programm 32
2.3 Starten des Programms 33
2.4 Ein Programm Schritt für Schritt anschauen ... 36

3. Amiga ein - LED aus 45

3.1 Wo liegt die LED im Amiga 45
3.2 Die LED wird ausgeschaltet 46
3.3 Zahlensysteme und Adressierungen 49
3.4 Namen statt Zahlen 53
3.5 Wir lassen die LED blinken 55
3.6 Wir erstellen ein Unterprogramm 57
3.7 Wir programmieren eine Warteschleife 59
3.8 Zusammenfassung 66

4. Weitere Anwendungen - die Maus 69

4.1 Abbruch auf Tastendruck 69
4.2 So stellt man Mausbewegungen fest 72

5. Das Betriebssystem hilft weiter 81

5.1 Wozu braucht man ein Betriebssystem 81
5.2 Offsets und Bibliotheken - was ist das 81
5.3 Eine Bibliothek ist immer da - Exec.library ... 83
5.4 Bibliothek öffnen und eine Funktion aufrufen 84

6. Wir öffnen ein eigenes Fenster **91**

6.1 So öffnet man ein DOS-Fenster 91
6.2 Die ersten Bildschirmausgaben - WRITE 97
6.3 Die ersten Tastatureingaben - READ 100
6.4 Wie kommt man an das aktuelle CLI-Fenster? 104
6.5 Programme ohne SEKA starten 106

7. Der Zugriff auf Disketten **109**

7.1 Dateien öffnen und lesen 109
7.2 Neue Dateien erstellen und schreiben 115
7.3 Einschub - So erhält man Zahlen von der Tastatur 122
7.4 Speicherbereiche auf Diskette sichern 127
7.5 So löscht man Dateien 138

8. Intuition **143**

8.1 Wir öffnen ein Intuition-Fenster 144
8.2 Wir überprüfen das Schließsymbol 151
8.3 Wie schreibt man Texte in ein Fenster? 157
8.4 Wir malen Punkte und Striche 162

9. Aufsteigen zum Amiga-Profi **169**

9.1 Wie nutzt man Informationen aus einem .... Intern 169
9.2 Wie setzt man Informationen für C um 173
9.3 Stopp - Spiele auf Tastendruck anhalten 181
9.4 Compare - Dateien vergleichen 186
9.5 Was man noch wissen sollte 194

10. Anhang **199**

10.1 Befehlsübersicht 199
10.2 Die Assembler 208
10.2.1 Der SEKA-Assembler 208
10.2.2 Der Profimat Assembler 217
10.2.3 Andere Assembler 221
10.3 Pannenhilfe 222
10.4 Lexikon 228

Stichwortverzeichnis **241**

Ein Mythos wird geknackt

Viele sagen, man müsse Maschinensprache beherrschen und damit programmieren, weil auf diese Weise Programme ca. hundertmal schneller werden als vergleichbare BASIC-Programme. Daß so mancher Programmierer für das Programm aber auch hundertmal so lange gebraucht hat, wird gerne verschwiegen.

Viele sagen, man müsse in Maschinensprache programmieren, weil dadurch Programme sehr viel kürzer als Programme in anderen Sprachen werden und vergessen, daß eine Diskette mit 880 KByte vielleicht fünf Mark kostet, also ein paar eingesparte KByte die Anstrengung kaum wert sind.

Wir dagegen möchten Sie in diesem Buch einfach und unkompliziert mit Maschinensprache vertraut machen. Sie werden dann schon selbst sehr schnell eigene Gründe finden, warum Ihnen Maschinensprache Spaß macht.

Damit der Spaß auch erhalten bleibt, und nicht durch Probleme in Frust umschlägt, haben wir uns einiges einfallen lassen:

- Alle Beispielprogramme in diesem Buch sind komplett abgedruckt. Sie brauchen also nicht erst noch verschiedene Teile zu einem Programm zusammenzusetzen, sondern finden immer das vollständige, lauffähige Programm.

- Wir liefern Ihnen eine Übersicht der wichtigsten Befehle des 68000. Dadurch können Sie schnell einmal nachschlagen, wenn Ihnen ein spezieller Befehl entfallen ist oder Sie für ein neues Problem einen Befehl suchen. Wir verzichten dabei bewußt auf die Aufnahme aller Befehle mit allen Möglichkeiten.

- Wir arbeiten in diesem Buch mit dem SEKA von Kuma. Alle für die Arbeit wichtigen Befehle und Vorgänge beschreiben wir ausführlich an den entsprechenden Stellen des Buches. Damit Sie aber jederzeit nachschlagen und nicht erst im Index suchen müssen, haben wir im Anhang eine Referenz des SEKA abgedruckt. Zusätzlich erläutern wir genau, was Sie bei der Nutzung des ebenfalls sehr beliebten Assemblers PROFIMAT wissen müssen. Sie können diesen also ebenfalls problemlos für die Beispiele verwenden.

- Damit Sie bei möglichen Problemen und Mißverständnissen eine schnelle Hilfestellung finden, haben wir in den Anhang ein spezielles Kapitel "Pannenhilfe" aufgenommen. Wenn also doch einmal etwas nicht so funktioniert, wie Sie sich das vorgestellt haben, finden Sie dort sicherlich eine Lösung.

- Sie werden in diesem Buch eine Reihe neuer Begriffe kennenlernen. Diese werden natürlich ausführlich und leichtverständlich erklärt. Sollte Ihnen aber in einem späteren Kapitel ein Begriff nicht mehr so ganz klar sein, schauen Sie doch einfach in unser Lexikon. Es befindet sich ebenfalls im Anhang des Buches.

Sie sehen also, wir haben uns viel einfallen lassen, um Sie problemlos und vor allem mit viel Spaß in die faszinierende Welt der Register, Taktzyklen, Bits und Bytes einzuführen. Vorhang auf für "Maschinensprache für Einsteiger"...

Münster, im Januar 1990 *Der Autor*

1. Ein paar notwendige Grundbegriffe

In diesem Kapitel wollen wir Ihnen einige grundsätzliche Begriffe erklären, die Sie zum Erlernen von Maschinensprache benötigen. Wir werden diese Begriffe im Verlauf der weiteren Kapitel in diesem Buch immer wieder verwenden. Daher ist es sinnvoll, sich vorher ein wenig mit ihnen vertraut zu machen.

Allerdings müssen Sie diese Grundbegriffe nicht vom ersten Augenblick an kennen. Wenn Sie sofort mit dem ersten Programm anfangen möchten, überschlagen Sie dieses Kapitel erst einmal, und beginnen Sie im nächsten Kapitel. Wenn Ihnen dann einmal ein Begriff unklar ist, lesen Sie sich einfach in Ruhe dieses Kapitel durch.

1.1 Von Bits, Bytes, Worten und Langworten...

Einige dieser Worte haben Sie bestimmt schon einmal gehört, doch was bedeuten sie eigentlich?

Was ist eigentlich ein Bit?

Ein Bit ist erst einmal die kleinste Informationseinheit in der Datenverarbeitung. Es gibt nur Aufschluß darüber, ob an einer bestimmten Stelle Strom fließt oder nicht. Denn nur diese beiden Zustände kann Ihr Computer unterscheiden. Alle Dinge, die sich in Ihrem Computer tun oder auch nicht tun, hängen nur von den unterschiedlichen Strömen ab, die in Ihrem Computer fließen. Wir definieren nun den Zustand, daß Strom fließt mit Null und den Zustand, daß kein Strom fließt, mit Eins. Da ein Bit also Auskunft darüber gibt, ob Strom fließt oder nicht, kann es nur die Werte 0 und 1 annehmen.

Was sind Bytes, Worte und Langworte?

Bytes, Worte und Langworte sind nun nicht mehr als eine ganz bestimmte Anzahl von Bits. So setzt sich ein Byte aus 8 Bits zusammen, ein Wort wiederum aus 2 Bytes bzw. 16 Bits. Die nächst größere Einheit sind die Langworte, die sich aus 2 Worten, dementsprechend 4 Bytes bzw. 32 Bits zusammensetzen.

1.2 Die unterschiedlichen Zahlensysteme

Um noch mehr über Bits, Bytes, Worte und Langworte zu erfahren, müssen wir erst über etwas anderes genauer Bescheid wissen. Wir wollen Ihnen im folgenden die verschiedenen Zahlensysteme erklären, die wir beim Umgang mit Maschinensprache kennen sollten.

Das Dezimalsystem

Das Zahlensystem, das Ihnen sicherlich am vertrautesten ist, ist das Dezimalsystem. Das Dezimalsystem ist das System, in dem wir üblicherweise denken und auch rechnen. Im Dezimalsystem haben wir, wie der Name sagt, zehn verschiedene Ziffern zur Verfügung. Des weiteren beruht das Dezimalsystem auf der Tatsache, daß mit jeder Stelle weiter nach links sich der Wert einer Ziffer verzehnfacht. Das Dezimalsystem ist somit ein System, das die Basis 10 besitzt.

Das Dualsystem

Nun gibt es aber noch andere Zahlensysteme. Eines davon, welches für uns auch von Wichtigkeit ist, ist das Dualsystem. Das Dualsystem besteht, wie der Name sagt, nur aus zwei Ziffern und zwar aus den Ziffern 0 und 1. Wird im Dualsystem eine Ziffer eine Stelle weiter nach links geschoben, so verdoppelt sich ihr Wert. Beim Dualsystem ist die Basis also 2.

Dieses Dualsystem, welches nur auf den Ziffern 0 und 1 beruht, eignet sich somit hervorragend zur Arbeit mit Bits, welche ja auch nur die Zustände 0 und 1 annehmen können. Um eine

duale Zahl von einer dezimalen Unterscheiden zu können, wird ihr ein Prozentzeichen "%" vorangestellt, also beispielsweise %10 = 2.

Das Hexadezimalsystem

Ein weiteres wichtiges Zahlensystem, für uns das wichtigste, ist das Hexadezimalsystem. Wie der Name schon sagt, ist es ein System, das auf der Basis 16 beruht. Wie sollen wir so ein System mit den uns zur Verfügung stehenden zehn Ziffern nun darstellen? Dieses ist in der Tat nicht ganz so einfach. Man hat sich hier mit Buchstaben geholfen. Buchstaben, werden Sie sagen, was haben die denn mit Zahlen zu tun? Nun wie schon gesagt, dient uns als Grundlage weiterhin unser Dezimalsystem. Wir können also jederzeit den Wert einer Zahl im Hexadezimalsystem in eine für uns vertraute Zahl des Dezimalsystems umrechnen.

Wie ist so ein Hexadezimalsystem aufgebaut? Im Hexadezimalsystem haben wir als die ersten Ziffern, genau wie im Dezimalsystem, die Ziffern 0 bis 9. Die weiteren Ziffern im Hexadezimalsystem sind A, B, C, D, E und F, wobei die Werte dieser Ziffern im Dezimalsystem folgenden Werten entsprechen.

A: 10
B: 11
C: 12
D: 13
E: 14
F: 15

Damit stehen uns also insgesamt 16 Ziffern zur Verfügung. Mit jeder Stelle, die eine Ziffer weiter links steht, versechzehnfacht sich also ihr Wert. Sie werden sich nun vielleicht fragen, warum wir noch ein drittes Zahlensystem benötigen, wo uns doch das dezimale so vertraut und das binäre so für Computer geeignet ist. Auch das Hexadezimalsystem hat einen ganz speziellen Vorteil. Ein Byte läßt sich darin nämlich mit zwei Ziffern darstellen (0 - FF), während im dezimalen Zahlensystem dazu drei Ziffern nötig wären (0 - 255). Außerdem kann man sich FFFF eben viel

leichter merken als 65535. Mit diesem Zahlensystem können also große Zahlen einfacher und anschaulicher dargestellt werden.

Hexadezimale Zahlen werden durch ein Dollarzeichen "$" gekennzeichnet. So ist beispielsweise $11 = 17.

Wie rechnet man Zahlen um?

Wir wollen Ihnen aber zunächst einmal ein paar Beispiele bringen, um Sie mit diesen Zahlensystemen vertrauter zu machen.

Die Umwandlung von Zahlen ist eigentlich ganz einfach, wenn man neben einem Taschenrechner einen kleinen Trick einsetzt. Man muß nämlich nur über eine Umrechnungstabelle der Zahlen von 0 - 15 verfügen, dann kann man durch geeignetes Aufteilen der Zahlen alle Probleme auf diese Tabelle zurückführen:

Dezimal	Hexadezimal	Binär
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Umrechnung Hex - Binär

Besonders einfach ist die Umwandlung zwischen hexadezimalen und dualen (binären) Zahlen. Genau vier Ziffern einer dualen Zahl entsprechen nämlich einer hexadezimalen Ziffer.

Beispiel 1: hex -> binär

Um beispielsweise die hexadezimale Zahl $9C in eine duale Zahl umzurechnen (man nennt duale Zahlen oft auch binär), schaut man für jede Ziffer der hexadezimalen Zahl in der Tabelle die vier Ziffern der dualen Zahl nach.

Hexadezimale Ziffer	9	C
Duale Ziffern	1001	1100

Somit ist $9C = %10011100

Beispiel 2: binär -> hex

Genauso einfach ist die umgekehrte Rechnung: Um eine binäre (also duale) Zahl in eine hexadezimale umzurechnen, faßt man jeweils vier Ziffern zusammen und schaut für diese die zugehörige hexadezimale Ziffer in der Tabelle nach. Um beispielsweise %00101010 umzuwandeln, bildet man Viererblöcke und findet:

Duale Ziffern	0010	1010
Hexadezimale Ziffer	2	A

Somit ergibt sich für %00101010 als hexadezimale Zahl $2A.

Umrechnung Hex – Dez

Am einfachsten gestaltet sich noch die Umwandlung von Hexzahlen in dezimale Zahlen. Dazu werden die hexadezimalen Ziffern in Dezimalzahlen umgewandelt und mit einem Vielfachen von 16 multipliziert.

Beispiel 1: hex -> dez

Um beispielsweise die hexadezimale Zahl $DA in eine dezimale Zahl zu verwandeln, ermittelt man die Ergebnisse für jede Ziffer, multipliziert und addiert zum Schluß:

Hexadezimal	D	A
Dezimal	13	10
Faktor	*16	*1

Somit ergibt sich 16*13 + 1*10 = 218

Beispiel 2: dez -> hex

Etwas schwieriger ist die Umwandlung von dezimalen in hexadezimale Zahlen. Am besten teilt man die Zahl so lange wie möglich durch 16 und notiert sich die Reste. Diese werden dann an Hand der Tabelle in hexadezimale Ziffern umgewandelt.

Um beispielsweise 199 umzurechnen, teilt man 199 durch 16 und erhält 12 Rest 7. Aus der Tabelle ermittelt man für 12 die Ziffer D und somit als Ergebnis $C7.

Beispiel 3: dez -> hex

An einem etwas umfangreicheren Beispiel wollen wir diesen Vorgang noch einmal durchprobieren. Wir wandeln die Zahl 1000 in eine Hexzahl um. Dazu teilen wir die 1000 so oft wie möglich durch 16 und merken uns die Anzahl der Teilungen:

```
1000: 16 = 62 Rest 8
62  : 16 =  3 Rest 14
```

Damit haben wir die erste Ziffer ermittelt, denn 3 läßt sich nicht weiter durch 16 teilen. Um die weiteren Ziffern zu bestimmen, multiplizieren wir die erhaltene 3 so oft mit 16, wie wir vorher geteilt haben, und benutzen als neuen Wert den Rest zur Ausgangszahl, also:

```
3*16*16 =  768
1000-768 = 232
```

Nun beginnt das Teilen durch 16 und merken erneut:

```
232:16 = 14 Rest 8
```

Da 14 sich nicht mehr durch 16 teilen läßt, ist unsere nächste Ziffer die zu 14 gehörende Ziffer aus der Tabelle, also E. Wir multiplizieren mit 16 und erhalten den Rest:

```
14*16   = 224
232-224 = 8
```

8 läßt sich überhaupt nicht mehr durch 16 teilen, also haben wir sofort die letzte Ziffer 8.

Damit ergibt sich 1000 = 3 * 16*16 + 14 * 16 + 8 * 1. Die Faktoren 3, 14 und 8 werden an Hand der Tabelle in die hexadezimalen Ziffern 3,E,8 umgewandelt, somit ergibt 1000 die Zahl $3E8.

Umrechnung Dez – Binär

Um binäre Zahlen in dezimale umzurechnen, multiplizieren Sie jede Ziffer der Binärzahl mit einem Vielfachen von 2 und addieren die Ergebnisse.

Beispiel 1:

Um %10010110 umzuwandeln, gehen Sie also folgendermaßen vor:

Binär	1	0	0	1	0	1	1	0
Faktor	128	64	32	16	8	4	2	1
Ergebnis	128	0	0	16	0	4	2	0

Somit ergibt sich 128+0+0+16+0+4+2+0 = 150.

Beispiel 2:

Grundsätzlich kann die Umrechnung von dezimalen in binäre Zahlen ähnlich erfolgen, wie das Umwandeln von dezimalen in hexadezimale Zahlen. Da hier aber viel häufiger geteilt werden

müßte, schlagen wir Ihnen einen einfacheren Weg (wenn auch einen kleinen Umweg) vor. Wandeln Sie die dezimale Zahl in eine hexadezimale Zahl um, diese läßt sich dann leicht in eine binäre Zahl umwandeln (siehe oben).

Um beispielsweise 199 umzurechnen (wir hatten das Beispiel oben schon einmal), teilt man 199 durch 16 und erhält 12 Rest 7. Aus der Tabelle ermittelt man für 12 die Ziffer D und somit als Ergebnis $C7.

Nun wandelt man von links nach rechts jeweil eine hexadezimale Ziffer an Hand der Tabelle in die vier binären Ziffern um:

Hexadezimale Ziffer	C	7
Binäre Ziffern	1100	0111

Das Ergebnis lautet also: 199 = $C7 = 11000111.

Zusammenfassung

Raucht Ihnen jetzt der Kopf? Dann sollten wir jetzt erst einmal eine kleine Verschnaufpause machen und die Ergebnisse zusammenfassen.

Jedes der drei Zahlensysteme hat seine Vorteile: Das binäre entspricht genau den Bits im Computer, mit dem hexadezimalen kann man auch größere Zahlen einfacher darstellen, und das dezimale Zahlensystem ist uns am vertrautesten. Zwischen den drei Zahlensystemen kann man alle Zahlen umwandeln, am einfachsten ist allerdings die Umwandlung zwischen hexadezimalem und dualem (binärem) System.

Am allerbesten ist aber, daß viele Assembler uns bei der Umrechnung helfen. Wir geben einfach die Zahl in dem von uns gewünschten Zahlensystem an (dezimal ohne besondere Kennzeichnung, hexadezimal mit "$" und dual mit "%"), und der Assembler weiß dann schon, was wir meinen.

Zusätzlich verfügt der SEKA-Assembler sogar über eine eingebaute Umwandlungsfunktion - wir kommen später noch auf dieses Hilfsmittel zurück. Totzdem sollte man die Zahlensysteme kennen und zwischen ihnen umrechnen können, nur so kann man das jeweils beste System wählen und bei Fehlern gegebenenfalls nachrechnen.

1.3 Der Speicher (RAM, ROM, WOM)

Fällt Ihnen etwas zum Thema Speicher ein? Vielleicht sofort die Antwort: Davon habe ich immer zu wenig. Nun, dabei können wir Ihnen leider auch nicht direkt helfen, da ist der Gang zum Händler mit dem notwendigen Geld in der Geldbörse unvermeidlich.

Um aber dem Amiga genau auf die Bits und Bytes gucken zu können, sollte man von seinem Speicher zweierlei wissen: Was für Speicher gibt es eigentlich, und was ist der Unterschied zwischen Inhalt und Adresse?

Art des Speichers

Am wichtigsten ist die Unterscheidung zwischen Speicher, den man ändern kann (**RAM**), und Speicher, der nicht geändert werden kann (**ROM**). Vielleicht fragen Sie sich jetzt, warum ein Amiga um alles in der Welt Speicher enthält, den man gar nicht ändern kann. Die Antwort ist einfach: Speicher, der geändert werden kann, verliert seine Informationen mit dem Ausschalten des Rechners. Hätte ein Amiga nur veränderbaren Speicher, wäre er nach dem Ausschalten "strohdoof", könnte also auch keine Informationen von einer Diskette in den Speicher lesen (Kickstart oder Workbench).

Daher muß zumindest ein Grundprogramm fest im Amiga gespeichert bleiben, das nach dem Einschalten abgearbeitet wird und das Lesen weiterer Informationen von Diskette oder Festplatte ermöglicht. Gerade diese Informationen stehen dann im nicht änderbaren Speicher, im ROM.

Hier besteht nun auch ein großer Unterschied zwischen dem Amiga 1000 und den beiden Kollegen 500 und 2000. Der Amiga 1000 hat wirklich nur dieses Grundprogramm im ROM, der Rest wird nach dem Starten von der Kickstartdiskette in den Speicher gelesen. Beim Amiga 500 und 2000 hat man dagegen richtig zugelangt und schon den größten Teil des Betriebssystems fest im ROM verankert. Dadurch entfällt das Laden von der Kickstartdiskette.

Gut, was ROM ist, wissen wir jetzt. RAM ist auch einigermaßen klar: Das ist der Teil des Speichers, in dem die Informationen geändert werden können, also damit auch der für uns interessante Teil des Speichers. Aber da wir nun schon mal beim Unterschied zwischen Amiga 1000 und 500/2000 sind, können wir auch noch das geheimnisvolle WOM abhaken.

Was ist denn eigentlich WOM?

Nun, die Anwort ist eigentlich einfach: WOM ist weder RAM noch ROM oder beides gleichzeitig. Alles klar? WOM ist die Abkürzung für "**W**rite **O**nce **M**emory" (zu deutsch: einmal beschreibbarer Speicher). Vor dem ersten Beschreiben ist er wie RAM, nach dem ersten Beschreiben wie ROM.

Am einfachsten kann man das erklären, wenn man sich zwei Probleme bei der Entwicklung des Amiga-Betriebssystems verdeutlicht:

1. Die Entwickler des Amiga wollten das Betriebssystem vor jeder Änderung schützen. Kein Programmierer sollte einfach Routinen des Betriebssystems manipulieren können und kein "abgestürztes", fehlerhaftes Programm sollte "ins" Betriebssystem schreiben können. Folglich mußte das Betriebssystem also ins ROM.

2. Unter dem Zeitdruck bei der Entwicklung des Amiga sollte zwar eine erste Version des Betriebssystems ausgeliefert werden (Version 1.0), aber diese sollte in der nächsten Zeit

noch verbessert und verfeinert werden. Da dies nun aber bei einem ROM nicht möglich ist, mußte das Betriebssystem ins RAM.

Damit war das Dilemma da: Am besten RAM und ROM gleichzeitig. Die Lösung war WOM. Grundsätzlich ist das natürlich beschreibbarer und veränderbarer Speicher, also RAM. Allerdings sorgten die Entwickler des Amiga dafür, daß man jeden Schreibzugriff, also jede Veränderung des Speichers sozusagen abschalten kann. Ist dies einmal erfolgt, verhält sich das WOM wie ein ROM, ist also nicht mehr änderbar. Dies gilt bis zu einem RESET oder dem Ausschalten des Amiga.

FAST- oder CHIP-Memory

Nun kennen wir also schon die wichtigsten Unterschiede im Speicher des Amiga, aber von einer letzten Unterscheidung sollte man doch noch wissen, sonst kann es einmal zu Problemen führen: dem Unterschied zwischen FAST-Memory und CHIP-Memory.

Um diesen Unterschied und die damit verbundenen Besonderheiten zu erklären, holen wir wieder ein wenig aus und beginnen beim Anfang, beim Amiga. Warum ist ein Amiga eigentlich so schnell? Anwort: Weil im Amiga nicht einer schuftet, sondern gleich mehrere. Neben dem eigentlichen Rechenkünstler, dem Prozessor, gibt es noch einige Spezialisten, die sogenannten Custom-Chips. Diese können einige spezielle Aufgaben erheblich schneller übernehmen als der Amiga. So kann etwa der Blitter superschnell Informationen im Speicher verschieben und verändern und eignet sich folglich hervorragend für grafische Aufgaben.

Klar ist auch, daß die Zusammenarbeit zwischen dem Prozessor und den Custom-Chips irgendwie geregelt sein muß. Diese Zusammenarbeit ist nicht so ganz einfach und daher wollen wir sie hier auch nicht erschöpfend behandeln, sondern nur das Wesentliche für den Unterschied zwischen CHIP- und FAST-Memory herausstellen. Prozessor und Custom-Chips wechseln sich bei der Arbeit mit dem Speicher ab.

Da die Custom-Chips nicht den gesamten Speicher des Amiga bearbeiten können, sondern (bei den bisherigen Modellen, da wird sich bald etwas ändern) nur die ersten 512 KByte, heißt dieser Speicher CHIP-Memory. Der Rest des Speichers kann von den Chips nicht bearbeitet werden. Auf diesen Teil des Speichers kann der Prozessor also ohne jede Arbeitsteilung und damit schneller zugreifen. Deshalb heißt der Speicher ab 512 KByte eben FAST-Memory.

Bleibt als letztes festzuhalten, was man beim Programmieren bezüglich des Unterschieds zwischen CHIP- und FAST-Memory beachten sollte:

1. Wenn ein Programm nur mit FAST-Memory arbeitet, ist es normalerweise etwas schneller als ein Programm im CHIP-Memory.

2. Wichtiger ist da schon der zweite Punkt: Für bestimmte Zwecke ist CHIP-Memory unersetzlich. Folglich sollte es auch nur verwendet werden, wenn dies notwendig ist.

3. Das Wichtigste ist aber: Alle Daten, die von den Custom-Chips bearbeitet werden sollen, müssen ins CHIP-Memory. Wenn also etwa der Blitter Daten verschieben soll und diese befinden sich im FAST-Memory, wird er sich trotzdem bemühen, allerdings wird er dann einen völlig falschen Speicherbereich verschieben. Und das Ergebnis einer solchen Operation sollte man sich eigentlich ersparen.

Wenn Sie sich nun fragen, warum viele Programme zwar auf einem Amiga 1000 funktionieren, aber nicht auf einem Amiga 500 oder 2000, dann haben Sie meist mit dem dritten Punkt die Antwort gefunden. Und wenn Sie sich außerdem fragen, wie Sie denn die Daten für die Custom-Chips ins CHIP-Memory befördern können, ist die Antwort auch ganz einfach: Speicher nimmt man sich nicht einfach, sondern läßt ihn sich vom Betriebssystem zuweisen. Und dabei gibt man eben zusätzlich an, daß es CHIP-Memory sein soll.

1.4 Die CPU - Das Wichtigste Ihres Computers

Wissen Sie, was ein Register ist? Oder haben Sie gestern noch jemandem den Unterschied zwischen einem 16- und einem 32-Bit-Datenbus erklärt? Fein, dann können Sie dieses Kapitel überfliegen oder beim Lesen prüfen, ob wir es denn auch richtig erklären. Für alle anderen wollen wir aber zumindest die wichtigsten Begriffe rund um einen Prozessor erläutern.

CPU - von MHz und Datenbussen

Sind 8 MHz achtmal schneller als 1 MHz?
Ist ein 32-Bit-Datenbus doppelt so breit wie 16 Bit?
Sind 80.286 mehr als 68.030 oder 6.502?

Der Kern des Amiga und auch der Teil, der uns im Laufe des Buches am meisten beschäftigen wird, ist der MC68000 von Motorola. Nun, mit den Namen von Prozessoren ist das so eine Sache: Warum der Prozessor 68000 heißt, ist uns auch nicht genau bekannt, vielleicht weil der Vorgänger MC6800 hieß und größere Zahlen gewöhnlich auch mehr Leistung versprechen. Aber lassen wir einmal beiseite, warum der Prozessor im Amiga 68000, im C64 6502 und in einem PC meist 8086 heißt.

Interessanter ist da schon die Tatsache, daß, wenn so ein Name erst einmal feststeht, Nachfolgemodelle meist eine größere Zahl zugewiesen bekommen:

> 6800, 68000, 68010, 68020, 68030 ...
> 8086, 80186, 80286, 80386 ...

Sie sehen, da gibt es ganz offensichtlich Gemeinsamkeiten. Nun gibt es einige wichtige Begriffe rund um einen Prozessor, die man zumindest einmal gehört haben sollte.

Taktfrequenz

Da in einem Amiga mehrere hochspezialisierte Kollegen (Prozessor, Blitter, Copper) zusammenarbeiten müssen, kann nicht jeder

einfach so schnell wie möglich arbeiten, sondern alle müssen sich aufeinander abstimmen. Das funktioniert so ähnlich wie auf einer Galeere aus längst vergangener Zeit: Nur haut dort einer auf die Pauke, und im Amiga gibt ein Quarz den Takt an. Übrigens hätte ein Galeerentrommler keine Chance im Amiga: Dort schlägt der Quarz die Trommel 7.160.000 mal pro Sekunde - ein ganz schöner Trommelwirbel.

Das Beispiel mit der Galeere hilft uns auch gleichzeitig, die Auswirkung der Taktfrequenz auf die Geschwindigkeit zu verstehen. So könnte unser Galeerentrommler zwar schneller auf die Pauke hauen, aber deshalb muß die Galeere noch nicht schneller fahren. Denn irgendwann fangen die armen Ruderer an, kleinere Ruderschläge zu machen. Aber auch bei gleicher Taktfrequenz müssen Computer nicht "gleich schnell" sein. Stellen Sie sich doch einfach mal vor, die eine Galeere hat ein paar Ruderer mehr oder kürzere Ruder.

Ein letzter interessanter Aspekt zur Taktfrequenz: Stellen Sie sich einmal vor, daß die Ruderer auf der Galeere nicht unbedingt bei jedem Trommelschlag einen Ruderschlag durchführen. Auf manchen Galeeren wird bei jedem zweiten Trommelschlag gerudert, auf anderen nur bei jedem dritten.

Für den Vergleich von verschiedenen Computern bedeutet das folgendes: Ein Prozessor versteht eine Reihe von Befehlen und kann diese verarbeiten. Allerdings schafft er diese selten innerhalb eines Taktes. Also hängt die Geschwindigkeit eines Prozessors nicht unbedingt von der Taktfrequenz ab, sondern auch davon, wieviele Takte er für bestimmte Befehle benötigt. Aber wer schneller trommelt, jagt zumindest der anderen Galeere Furcht ein. Also: Keine Angst vor "schnellen" Konkurrenten.

Was tut ein Bus im Computer

Stellen Sie sich einmal vor, da sitzt ein einsamer MC68888 auf der Platine Ihres Amiga, und am anderen Ende sitzen die Speicherbausteine. Da kann der Prozessor lange trommeln, passieren wird nichts. Vielmehr müssen die einzelnen Elemente eines Computers irgendwie mit Leitungen verbunden sein.

Die Anzahl dieser Leitungen ist kein Zufall und bestimmt gleichzeitig die Geschwindigkeit des Rechners. Hier gilt wirklich: 16 Leitungen können schneller Informationen transportieren als 8. Die zusammengehörenden Leitungen nennt man in der Computersprache Bus.

Der Amiga hat sogar gleich zwei dieser Busse: einen Daten- und einen Adreßbus. Auf dem Adreßbus kann der 68000 festlegen, welche Speicheradresse er bearbeiten will. Der Inhalt dieser Adresse kann dann über den Datenbus gelesen oder verändert werden.

Da wir uns im Inneren des Amiga nun schon recht gut auskennen, können wir uns auch gleich noch Anzahl der Leitungen dieser Busse anschauen: Der Adreßbus hat 23 Leitungen und der Datenbus 16. Mit 16 Datenleitungen kann man Zahlen von 0 bis 2^16 übertragen, also 0 - 65536. Anschaulicher wird die Zahl, wenn man sie geschickt aufteilt: 2^16 = 2^8 * 2^8. Da 2^8 ein Byte ergibt, sind dies gerade zwei Byte oder ein Wort.

Bleibt noch die Frage, wie viele Adressen der MC68000 denn verarbeiten kann? Mit 23 Adreßleitungen können 2^23 Adressen gebildet werden, also 8.388.608. Da aber jeweils Worte, also zwei Bytes bearbeitet werden (der Datenbus ist ja 16 Bit breit), muß die Zahl noch mit zwei multipliziert werden und ergibt 16777216. Bekannter ist diese Zahl als 16 Megabytes, also ist im Amiga grundsätzlich Platz für 16 MByte Speicher.

Wenn Sie nun auf die Anzeige des Speichers in der Workbench-Titelzeile schauen, werden Sie leicht feststellen, daß im Amiga sozusagen noch gähnende Leere herrscht. Denn von den möglichen 16 Megabytes sind im Amiga 500 standardmäßig nur 1/2 Megabyte und im Amiga 2000 gerade mal ein Megabyte vorhanden.

Von Registern, Flags und Stacks

Nachdem wir nun wissen, wie schnell ein 68000 ist und wie er mit der restlichen Welt im Amiga verbunden ist, wollen wir

noch einen Blick unter sein Gehäuse wagen. Dort gibt es noch einmal reichlich interessante Details zu sehen.

Prozessor oder CPU?

Haben Sie schon einmal den Begriff CPU gehört? Und fragen Sie sich vielleicht schon die ganze Zeit, was den der Unterschied zwischen dem Prozessor und der CPU ist? Nun, ganz einfach: keiner. Im Amiga sitzt der 68000-Prozessor und, weil er die meiste Arbeit macht, heißt er CPU (Central Processing Unit = Zentrale Verarbeitungseinheit). Und eine seiner wichtigsten Fähigkeiten ist das Rechnen. Aber keine Sorge: So viel kann der 68000 eigentlich gar nicht. Seine Rechenfähigkeiten liegen etwa auf dem Niveau der Grundschule also Addieren, Subtrahieren, Multiplizieren und Dividieren.

Was sind eigentlich Register?

Haben Sie schon einmal jemand zugeschaut, wie er 16 * 23 im Kopf rechnet und dabei die Hände zur Hilfe nimmt. Dann wissen Sie in etwa auch, wozu ein Prozessor Register benötigt. So ganz paßt das Bild aber nicht, denn der Prozessor bewahrt alle Zahlen, mit denen er rechnet, in den Registern auf und nicht nur die Zwischenergebnisse und benötigt die Register noch für einiges mehr.

Da auch im 68000 wie überall in einem Computer alles hochspezialisiert ist, sind auch die Register eines Prozessors in drei Gruppen aufgeteilt: acht Datenregister, acht Adreßregister und einige Spezialregister. Während wir auf einige Spezialregister weiter unten eingehen, soll hier kurz auf den wichtigsten Unterschied zwischen den ersten beiden Registertypen hingewiesen werden:

Datenregister Hier bewahrt der Prozessor die Zahlen auf, mit denen er rechnen will.

Adreßregister Mit den Adreßregistern merkt sich der 68000 wichtige Adressen von Zahlen oder Zahlengruppen.

Jedes dieser Register enthält übrigen 32 Bit und kann also Zahlen zwischen 0 und 2^32 verarbeiten. 2^32 ist die gigantische Zahl von 4.294.836.225 oder besser 4 Gigabyte beziehungsweise 4000 Megabyte.

So merkt der Prozessor Ergebnisse - Flags

Wenn der 68000 rechnet, können einige Besonderheiten passieren, die man als Programmierer gerne erfahren will. Diese werden vom Prozessor in einem speziellen Register, dem Statusregister, notiert. Dazu werden einfach einige der 16 Bits im Statusregister verwendet. Jedes hat eine spezielle Bedeutung, und wenn das zugehörige Ereignis eintritt, wird das Bit auf 1 gesetzt, ansonsten auf 0.

Wird beispielsweise das Ergebnis einer Subtraktion negativ, so wird das Negativ-Flag gesetzt. Dies ist per Definition einfach das vierte Bit im Statusregister. Wenn Sie also später einmal in einem selbstgeschriebenen Maschinenprogramm Ihre Ausgaben von den Einnahmen abziehen, fragen Sie den Prozessor anschließend besser nach dem Negativ-Flag. Wenn er Ihnen dann mitteilt: "Ist gesetzt", sollten Sie bald etwas dagegen tun: Senken Sie Ihr Defizit.

Ein PC im 68000

Keine Sorge, Sie haben keinen Personalcomputer in Ihrem Amiga. PC steht als Abkürzung für Programcounter oder deutsch Programmzähler. Ein Maschinenprogramm wird im Amiga irgendwo im Speicher abgelegt und Befehl für Befehl bearbeitet. Damit der Prozessor aber immer genau Bescheid weiß, wo er gerade ist und was denn der nächste Befehl ist, hat er eben diesen PC.

Der PC ist also so etwas ähnliches wie in BASIC die aktuelle Zeile. Und weil der Vergleich so schön ist kann man auch gleich fragen, ob der Prozessor denn auch an einer beliebigen Stelle mit dem Programm fortfahren kann wie beim GOTO in Basic. Antwort: Er kann, und zwar GOTO und GOSUB.

Das Kurzzeitgedächtnis der CPU - Der Stack

Greifen wir das GOSUB noch einmal kurz auf. In BASIC bedeutet das: Unterbrich die Arbeit an der Stelle, mach an der angegebenen Stelle weiter, bis der Befehl RETURN kommt, und nimm dann die Arbeit an der alten Stelle wieder auf. Wie sich dabei BASIC die alte Stelle merkt, soll an dieser Stelle nicht weiter interessieren, aber auch der 68000 muß sich natürlich die Adresse merken, an der die Arbeit unterbrochen wurde. Dazu dient ein spezieller Speicherbereich, der sogenannte Stack.

Wenn Sie sich fragen, warum das nicht viel besser ein Stackregister erledigen könnte, dann ist die Antwort einfach. Der Prozessor könnte immer nur ein einziges Unterprogramm aufrufen, weil er sich nur eine Rücksprungadresse merken könnte. Folglich wird das etwas anders gelöst. Der Prozessor merkt sich in einem speziellen Stackregister nicht die Rücksprungadresse, sondern den Beginn des speziellen Speichers, des Stacks. Sobald er auf den Befehl "GOSUB" trifft, setzt das Stackregister auf die nächste freie Adresse im Stack und speichert die Rücksprungadresse dort ab. Dieser Vorgang kann also viele Male hintereinander ablaufen.

Ist dann der Rücksprung fällig, holt sich der Prozessor über die Adresse im Stackregister die Rücksprungadresse. Wurden mehrere GOSUB hintereinander ausgeführt, holt er sich eben nacheinander alle vorher gespeicherten Adressen. Weil das Stackregister immer auf die aktuell gültige Rücksprungadresse "zeigt", heißt das Stackregister Stackpointer (SP), von engl. to point = zeigen.

Hinweis: Der Prozessor 68000 benutzt als Stackregister immer das Adreßregister A7. Wenn Sie also während unserer Beispiele einmal etwas ausprobieren und Veränderungen vornehmen wollen, dürfen Sie keinesfalls einfach dieses Register verwenden, weil der Prozessor sonst nicht mehr die Rücksprungadresse finden kann.

2. Es geht los - ein erstes Programm

Nun solls aber endlich losgehen mit dem ersten Programm. Dabei werden wir nicht nur die ersten Befehle des 68000 kennenlernen, sondern uns auch mit dem Hilfsmittel SEKA beschäftigen und dabei vieles wiederfinden, was im letzten Kapitel angesprochen wurde.

2.1 Wir addieren Zahlen

Bevor wir unser erstes Programm erstellen können, müssen wir natürlich erst einmal unseren Assembler starten. Wir werden in diesem Buch alle Programme anhand des SEKA-Assemblers erklären und eingeben. Sollten Sie einen anderen Assembler besitzen, schauen Sie bitte im Anhang nach. Dort werden Ihnen die Unterschiede, die Sie beachten müssen, erklärt.

Starten des SEKA

Um den SEKA-Assembler zu starten, legen Sie die Diskette mit dem Assembler in Laufwerk DF0 ein und starten den Assembler mit der Eingabe von:

```
DF0:SEKA
```

Bestätigen Sie mit der Eingabe-Taste. Dieses müssen Sie grundsätzlich tun, wenn Sie sich im Befehls-Modus befinden und dort etwas eingeben. Wir werden dies im folgenden nicht immer ausdrücklich erwähnen. Nun erscheint ein neues Fenster auf Ihrem AMIGA. Als erstes müssen Sie eine Angabe darüber machen, wieviel Speicher sich der Assembler reservieren soll. Für sinnvoll halten wir es, uns 100 KByte Speicher reservieren zu lassen. Geben Sie also 100 ein:

```
WORKSPACE KB> 100
```

Hinweis: Wenn Sie einen Amiga 500 haben, können Sie auch einen kleineren Wert - beispielsweise 30 - angeben.

Damit haben Sie alle Vorbereitungen für die Arbeit mit dem Assembler getroffen und können nun direkt mit dem ersten Programm beginnen.

Das erste Programm im Editor

Nach dem Start des SEKA-Assemblers befinden Sie sich zunächst einmal im Befehls-Modus. In diesem Modus können Sie Programme assemblieren (also in ausführbare Programme umwandeln lassen), starten usw. Um ein Programm zu schreiben, müssen Sie in den Editor wechseln. Dazu drücken Sie die ESC-Taste. Ihr Bildschirm teilt sich nun, und in der oberen Hälfte sehen Sie:

```
1 <END>
```

Der Cursor befindet sich in dieser Zeile und zwar auf der spitzen Klammer vor dem END. Drücken Sie einfach ein paarmal die Eingabe-Taste, und es erscheinen neue Zeilen auf dem Bildschirm, die durchnumeriert sind. Das <END> steht immer in der letzten Zeile und kann auch nicht gelöscht oder verändert werden.

Wir wollen sofort das erste kleine Programm in den Editor schreiben, es soll zwei Zahlen addieren. Nehmen wir doch einfach die Zahlen 5 und 13. Dazu gehen Sie folgendermaßen vor:

Bewegen Sie den Cursor in die erste Zeile, und schreiben Sie folgendes Programm in den Editor:

```
START:
MOVE.L #$05,D0
MOVE.L #$0D,D1
ADD.L D0,D1
Illegal
```

Hinweis: Drücken Sie bitte am Ende jeder Zeile die Eingabe-Taste. Das Zeichen "#" ist neben der Return-Taste.

In diesem Programm kommen zwei Befehle vor, die eine Erklärung erfordern. Nehmen wir zunächst den Befehl MOVE.

Der Befehl MOVE

Dieser Befehl kommt aus dem Englischen und bedeutet übersetzt "Bewegen". Damit kommen wir der Erklärung dieses Befehls schon ganz schön nahe, denn er macht wirklich so etwas, was man mit bewegen bezeichnen könnte. Die genauere Umschreibung ist vielleicht "Übertragen". Der Befehl überträgt also etwas und zwar Zahlen.

Hinter dem Befehl steht noch ein ".L". Hiermit wird angegeben, das es sich bei dem zu übertragenden Wert um ein Langwort handelt. Ist ein Wort zu übertragen, wird als Anhängsel ein ".W" benutzt, handelt es sich um ein Byte, wird ein ".B" angehangen. Geben Sie gar kein Anhängsel an, wird immer ein "Wort" übertragen. Sie erinnern sich vielleicht an die Erklärung von Langworten, Worten und Byte. In Kapitel 1.1 wird auf diese Unterschiede näher eingegangen.

Diese Anhängsel werden nicht nur für den Befehl MOVE benutzt, sondern noch für eine Menge anderer Befehle, wie Sie im folgenden noch sehen werden.

Hinter dem Befehl sehen Sie noch ein paar Zeichen und zwar einmal ein "#" und danach ein "$". Das Zeichen "#" bedeutet, daß es sich bei der folgenden Angabe um den Wert einer Zahl handelt und nicht um den Inhalt einer Speicherstelle. Das "$" sagt aus, daß der zu übertragende Wert hexadezimal angegeben ist, das kennen Sie ja schon.

Jetzt können Sie sich sicher auch vorstellen, was die "05" und das "0D" hinter diesen Zeichen für eine Bedeutung haben. Es ist die hexadezimale Schreibweise der Zahlen "5" und "13".

Hiermit sind alle Informationen für den zu übertragenden Wert vorhanden. Nun fehlt nur noch die Angabe, wohin er übertragen werden soll. Diese Angabe erfolgt hinter dem Komma. Bei uns

handelt es sich hier um die Datenregister D0 und D1. Dies soll an dieser Stelle als Erklärung für den Befehl MOVE zunächst einmal genügen.

Der Befehl ADD

Dieser Befehl wird zum Addieren benötigt.

Das ".L" bedeutet, wie schon bei MOVE, daß Langworte behandelt werden, also vier Bytes auf einmal.

Die nun folgenden Angaben geben Auskunft über das, was addiert werden soll. In unserem Fall handelt es sich um die Inhalte der beiden Datenregister D0 und D1, in die wir unsere zu addierenden Werte eingegeben haben. Das Ergebnis einer solchen Addition wird immer in dem Register abgelegt, das hinter dem Komma steht. Bei uns würde das Ergebnis also im Datenregister D1 abgelegt werden.

Hinweis: Dies gilt sogar ganz allgemein. Vor dem Komma steht immer die Quelle und hinter dem Komma immer das Ziel einer Operation.

2.2 Wir assemblieren das Programm

Nun wollen wir unsere Programmzeilen vom Assembler in Maschinensprache übersetzen lassen - diesen Vorgang nennt man assemblieren.

Assemblieren des Programms

Als nächstes müssen wir unser kleines Programm assemblieren, d.h. in ein lauffähiges Maschinenspracheprogramm übersetzen. Dazu wechseln wir in den Befehls-Modus. Drücken Sie dazu die ESC-Taste. Um nun das Programm zu assemblieren, geben Sie ein "A" für "Assemblieren" ein und drücken anschließend die Eingabe-Taste. Auf dem Bildschirm werden Sie nach Optionen

gefragt, die beim Assemblieren berücksichtigt werden sollen. Wir wollen hier keine Option eingeben und drücken daher einfach die Eingabe-Taste.

Haben Sie bei der Eingabe des Programms keine Fehler gemacht, erscheint die Meldung:

```
NO ERRORS
```

Bei Übersetzungsschwierigkeiten bieten wir Ihnen die perfekte Übersetzung: keine Fehler.

Sollte hier eine andere Meldung erscheinen, sehen Sie sich unser Programm noch einmal genau an und vergleichen es mit dem, was in Ihrem Editor steht. Verbessern Sie gegebenenfalls die Fehler, und assemblieren Sie es neu.

Auf die Fehlermeldungen, die auftreten können, wollen wir an dieser Stelle nicht eingehen. Möchten Sie jedoch etwas darüber in Erfahrung bringen, so schauen Sie sich im Anhang die Pannenhilfe an.

2.3 Starten des Programms

Jetzt wollen wir dieses kleine Programm starten bzw. abarbeiten lassen.

Dazu geben Sie ein:

```
G
```

Dieses "G" steht für "GO", was soviel wie "START" bedeutet. Nach der Eingabe drücken Sie die Eingabe-Taste.

Danach können Sie eine Angabe über die von Ihnen gewünschten Breakpoints machen, das sind sogenannte Abbruch-Punkte, die manchmal sehr nützlich sind, wenn man auf der Suche nach Fehlern in einem Programm ist. Wir wollen an dieser Stelle nicht

weiter darauf eingehen. Da wir keine Breakpoints setzen wollen - oder wollen Sie dieses kurze Programm auch noch unterbrechen -, drücken wir zum Starten des Programms die Eingabe-Taste.

Wo steht das Ergebnis

Auf dem Bildschirm erscheint nun ein Block mit Informationen. Dieser Block sieht in etwa so aus:

```
***Illegal Instruction at $C44418
D0=00000005 00000012 00000000 00000000 00000000 00000000 00000000 00000000
A0=00000000 00000000 00000000 00000000 00000000 00000000 00000000 00C3AF94
SSP=00C3BF34 USP=00C3AF94 SR=0000 PC=C44418 ILLEGAL
```

Hinweis: Wundern Sie sich nicht, wenn bei Ihnen an einigen Stellen andere Zahlen erscheinen, beim AMIGA ist alles "relativ".

Sie werden sich sicher fragen, was hier überhaupt passiert ist, und was dieser Block bedeutet. Es ist alles wahnsinnig schnell gegangen, und Sie können auf den ersten Blick kein Ergebnis Ihrer Addition erkennen.

Und doch steht das Ergebnis der Addition schon in dem Block, der nach dem Starten des Programms erschienen ist. Kommen wir also zur Erklärung dieses Blockes, der sogenannten Registeranzeige.

In der ersten Zeile wird Ihnen mitgeteilt, an welcher Stelle bzw. nach welchem Befehl Ihr Computer das Programm verlassen hat.

In unserem Fall führte anscheinend das ILLEGAL zum Abbrechen des Additions-Programms.

Das ILLEGAL

Der ILLEGAL-Befehl dient nur dazu, ein Programm zu beenden. Der Assembler übersetzt das ILLEGAL in einen nicht erlaubten Maschinensprachebefehl. Gelangt das Programm an diese Stelle, unterbricht der Prozessor die weitere Arbeit, und der

SEKA erhält wieder die Kontrolle. Dieser Befehl ist für unser Programm sehr wichtig, damit es an dieser Stelle beendet wird, und es nicht zu Operationen kommen kann, die sich der Programmierer nicht wünscht und auf die er keinen Einfluß hat. Trifft ein Programm auf diesen Befehl, so erscheint bei der Ausgabe des Informationsblockes (der Registeranzeige) immer in der ersten Zeile:

```
***Illegal Instruction at $C44418
```

Nur die Angabe der Speicherstelle wird sich unterscheiden.

In der nächsten Zeile stehen die Inhalte der Datenregister D0 bis D7, in der darunterstehenden Zeile die Inhalte der Adreßregister A0 bis A7.

Die letzte Zeile gibt Informationen über weitere spezielle Register, auf deren Besonderheiten wir hier nicht weiter eingehen wollen. Am Ende der letzten Zeile wird Ihnen noch mitgeteilt, welche Operation Ihr Prozessor als nächstes ausführen würde.

Das Ergebnis

Für uns sind im Moment nur die Inhalte der Datenregister von Interesse, denn hier steht auch das Ergebnis unserer Addition. Wie schon bei der Erklärung des Befehls ADD beschrieben, wird das Ergebnis einer Addition immer in dem Datenregister, das hinter dem Komma steht, abgelegt. In unserem Fall war dies das Datenregister D1. Schauen wir uns also die Zeile, in der der Inhalt der Datenregister steht noch einmal genau an.

In dieser Zeile sind acht Spalten zu erkennen, die jeweils aus acht Zeichen bestehen. Diese acht Spalten entsprechen den acht Datenregistern, die es gibt. Die erste Spalte gibt den Inhalt des Datenregister D0 an, die zweite Spalte den Inhalt des Datenregisters D1 usw. Das Ergebnis unserer Addition müßte somit in der zweiten Spalte der Datenregisterzeile stehen. Diese hat den Inhalt:

```
00000012
```

Inhalte von Registern werden immer hexadezimal ausgegeben. Dezimal hat unser Ergebnis also den Wert:

```
1*16 + 2*1 = 18
```

Und das scheint ja auch wohl richtig zu sein, denn dieses Ergebnis haben Sie sich bestimmt schon selbst errechnet und somit wohl auch erwartet.

2.4 Ein Programm Schritt für Schritt anschauen

Doch schauen wir uns einmal genau an, was überhaupt passiert ist, bzw. was unser Programm im einzelnen gemacht hat. Dafür müssen wir unser Programm neu starten. Aber Vorsicht, wenn Sie Ihr Programm nun einfach neu mit dem Befehl G starten, kann es passieren, daß es nicht richtig abgearbeitet wird. Woran liegt dies nun?

Der Adreßzähler PC

Um dieses zu erklären, müssen wir ein wenig auf die Arbeitsweise des Prozessors im AMIGA eingehen. Damit der SEKA ein Programm starten kann, muß er zunächst einmal wissen, an welcher Stelle des Speichers sich dieses Programm befindet, wo es also im Speicher abgelegt wurde. Danach muß dem Prozessor mitgeteilt werden, daß er an dieser Stelle anfangen soll, die Maschinenbefehle abzuarbeiten. Dies geschieht mit Hilfe des Adreßzählers oder abgekürzt PC für Programcounter.

Hat der Prozessor ein Programm einmal abgearbeitet, müssen ihm all diese Sachen noch einmal neu mitgeteilt werden. Dieses geschieht automatisch durch den SEKA beim Assemblieren. Um also ein Programm erneut zu starten, muß es vorher neu assembliert werden. Später werden wir andere Möglichkeiten kennenlernen, ein Programm mehrmals zu starten.

Sie können sich auch ganz einfach anschauen, wo der SEKA Ihr Programm abgelegt hat und an welcher Stelle sich der Adreßzähler zur Zeit befindet.

Wo beginnt das Programm im Speicher

Um sich anzuschauen, wo sich der Anfang Ihres Programm befindet, geben Sie folgende Zeile ein:

```
?START
```

Wir geben hier START, weil wir das als als Label für den Start unseres Programms gesetzt haben. Ein Label ist sozusagen ein Merker, durch den man bestimmten Stellen im Programm einen Namen geben kann.

Danach erscheint auf Ihrem Bildschirm der Wert, an dem der SEKA den Anfang Ihres Programms abgelegt hat. Dieser Wert wird zunächst hexadezimal ausgegeben, Sie können dieses an dem "$" vor dem Wert erkennen, dahinter steht noch der dezimale Wert.

Mit dem "?" können Sie sich also für Ausdrücke oder Labels anzeigen lassen, wo sich diese nach dem Assemblieren im AMIGA befinden. Daher ist es auch wichtig, Programme immer mit einem Label zu beginnen, in unserem Beispielprogramm das Label "START". Dann können wir jederzeit die Startadresse unseres Programms erfahren.

Wo befindet sich der PC im Augenblick

Wie können Sie nun feststellen, an welcher Stelle sich der Adreßzähler zur Zeit befindet, welcher Befehl also als nächstes ausgeführt würde? Geben Sie ein:

```
XPC
```

PC ist hierbei der Name des Adreßzählers (Sie erinnern sich: PC = Programcounter). Mit "X" können Sie sich den Wert von Regi-

stern ausgeben lassen und sie ändern, in unserem Fall den Wert des Adreßzählers, der ja auch ein Register ist, und diese Werte ändern.

Sind diese beiden Werte unterschiedlich, enthält der PC also nicht die Startadresse des Programms, so kann Ihr Programm natürlich nicht vernünftig gestartet und abgearbeitet werden. Um den Adreßzähler PC auf den richtigen Wert (START = $********) zu setzen, sollten wir das Programm neu assemblieren. Dazu verlassen Sie die Anzeige des PC mit <Return> und geben im Befehls-Modus ein:

```
A
```

Wieder werden Sie nach den Optionen gefragt, drücken Sie die Eingabe-Taste, denn wir wollen keine Optionen vorgeben.

Step bei Step

Diesmal wollen wir uns nicht nur einfach das Ergebnis anschauen, sondern wir wollen dem Prozessor einmal bei der Arbeit zusehen.

Dazu lassen wir unser Programm Schritt für Schritt abarbeiten und sehen uns nach jedem Schritt an, was geschehen ist. Um sich ein Programm so anzuschauen, müssen Sie nach dem Assemblieren eingeben:

```
S
```

Dadurch wird vom Prozessor genau ein Befehl des Programms abgearbeitet. Nach Drücken der Eingabe-Taste erscheint wieder ein Block auf dem Bildschirm, der dem vorher beschriebenen im Aussehen gleicht. Es fehlt zwar die erste Zeile, was daran liegt, daß unser Programm zu diesem Zeitpunkt noch nicht auf ein ILLEGAL gestoßen ist, und die Inhalte der Speicherstellen unterscheiden sich von dem oben erklärten Block, des weiteren das Ende der letzten Zeile, in der ja immer steht, welchen Befehl der Prozessor als nächstes ausführt.

Schauen wir uns diesen Befehl einmal an. Hier steht:

```
MOVE.L #$0000000D,D1
```

Dieses ist nun aber die zweite Befehlszeile, die wir eingegeben haben, die erste scheint somit schon abgearbeitet zu sein. Allerdings unterscheidet sich die ausgegebene Programmzeile von der Zeile, die wir oben im Editor eingegeben hatten. Diese lautete nämlich:

```
MOVE.L #$0D,D1
```

Wie können Werte eingegeben werden und wie werden sie gezeigt

Wir haben uns hier einer einfacheren Schreibweise bedient. Denn es ist wohl etwas mühsam, die ganzen Nullen einzugeben und sich dabei nicht zu verzählen. Um also einen Wert einzugeben, können wir die Nullen vor diesem Wert weglassen. Wichtig ist es anzugeben, ob es sich bei dem zu übertragenden Wert um ein Byte, Wort oder Langwort handelt. Es kann nämlich vorkommen, das sich in der Speicherstelle, in die der Wert übertragen werden soll noch etwas befindet. Würden wir dann nur ein Wort übertragen, also zwei Bytes, so würde der Prozessor nur die letzten vier Stellen dieser Speicherstelle ändern, die ersten vier würden bestehen bleiben. Es stände dann in dieser Speicherstelle eventuell etwas ganz anderes als das, was wir wollen.

Der Grund, weshalb wir doch noch eine Null geschrieben haben, ist ganz einfach der, daß es üblich ist, Eingaben mindestens in Byte zu machen. Der Assembler gibt uns jedoch immer den vollständigen Wert mit acht Ziffern in hexadezimaler Schreibweise aus.

So, dazu nun erst einmal genug. Wir sehen also, daß der erste Befehl, den wir eingegeben haben, schon abgearbeitet wurde. Wir aber wollten dem Prozessor bei der Arbeit zusehen und zwar von Anfang an. Wie können wir dieses erreichen?

Wir wollen den Anfang des Programms sehen

Gehen Sie mit <Esc> noch einmal in den Editor, und geben Sie hinter

```
START:
```

eine neue Zeile ein. Dazu bewegen Sie zunächst den Cursor hinter den Doppelpunkt und drücken die Eingabe-Taste. Dadurch erhalten Sie eine neue leere Zeile. Hier geben Sie nun ein:

```
NOP
```

Was hat dieser Befehl zu bedeuten?

Der Befehl NOP

Das NOP steht für "NO OPERATION", übersetzt also "keine Tätigkeit". Und ganz genau das macht Ihr Prozessor, wenn er auf diesen Befehl trifft, nämlich "NICHTS". Dieser Befehl scheint somit überflüssig zu sein, doch schon in unserem Beispiel werden wir sehen, das er durchaus eine Funktion erfüllt. Uns gelingt es mit Hilfe dieses Befehls, dem Prozessor von Anfang an bei der Arbeit zuzuschauen.

Das gesamte Programm im Einzelschritt anschauen

Aktivieren Sie wieder mit <Esc> den Befehls-Modus, und assemblieren Sie neu. Jetzt können wir uns die Arbeit des Prozessors von Anfang an Schritt für Schritt ansehen.

Geben Sie erneut "S" ein, und drücken Sie die Eingabe-Taste. Wieder erscheint der schon bekannte Block. Schauen wir uns das Ende der letzten Zeile an. Hier steht:

```
MOVE.L #$00000005,D0
```

Dies ist der nächste Befehl, den der Prozessor ausführen wird. Der nächste, weil er, ohne daß wir es bemerkt haben, schon den Befehl NOP ausgeführt hat, dieser aber eben dazu führt, das keine Tätigkeit ausgeführt wird.

Der Prozessor soll den Wert "5" ins Datenregister "D0" übertragen. Wir wollen überprüfen, ob dieses wirklich geschieht. Dazu geben Sie im Befehls-Modus erneut ein "S" ein. Nach Bestätigung mit der Eingabe-Taste erscheint ein Informationsblock mit der Ausgabe der einzelnen Register auf Ihrem Bildschirm. Diesen wollen wir uns etwas genauer ansehen.

```
D0=00000005 00000012 00000000 00000000 00000000 00000000 00000000 00000000
A0=00000000 00000000 00000000 00000000 00000000 00000000 00000000 00C39BFC
SSP=00C3AB9C USP=00C39BFC SR=8000 t PC=C4444C MOVE.L #$0000000D,D1
```

Als erstes wollen wir überprüfen, ob im Datenregister D0 wirklich der Wert "5" steht. Dazu müssen wir uns die erste Zeile der Registeranzeige ansehen und dort die erste Spalte, denn hier steht ja, wie schon beschrieben, der Inhalt des Datenregisters D0. Die erste Spalte der ersten Zeile hat den Inhalt :

```
00000005
```

Sie sehen also, der Wert "5" ist in das Datenregister D0 übertragen worden. Des weiteren interessiert uns nun noch, welcher Befehl als nächstes abgearbeitet wird. Dazu schauen wir uns das Ende der letzten Zeile an. Hier steht, wie Sie sicher schon erwartet haben, der Befehl:

```
MOVE.L #$0000000D,D1
```

Dies ist in unserem Programm der nächste Befehl, den der Prozessor abarbeiten muß. Er soll dem Datenregister D1 den Wert "D", der dezimal dem Wert "13" entspricht, zuweisen. Um unser Programm nun weiter zu verfolgen, geben Sie erneut ein "S" ein. Nach Bestätigung mit der Eingabe-Taste erscheint wieder eine Registeranzeige. Hier sollte in der zweiten Spalte der ersten Zeile folgendes stehen:

```
0000000D
```

Das Datenregister D1 enthält somit den Wert "D". Des weiteren sollte in der unteren Zeile am Ende stehen:

```
ADD.L D0,D1
```

Die Inhalte der beiden Register D0 und D1 sollen also addiert werden. Schauen wir uns den nächsten Schritt an, in dem wir erneut ein "S" eingeben und mit der Eingabe-Taste bestätigen.

Interessant ist für uns der Inhalt des Datenregisters D1, in dem sich nun das Ergebnis der Addition befinden sollte. Schauen wir uns also die zweite Spalte der ersten Zeile an, hier steht wie wohl auch erwartet:

```
00000012
```

In der letzten Zeile steht am Ende noch unser ILLEGAL, mit dem wir beim Prozessor einen Abbruch erzwingen. Geben Sie also ein letztes Mal ein "S" ein, und wiederum erscheint ein neuer Block. Dieser sollte nun dem Block, den wir erhalten haben, als wir unser Programm durchlaufen lassen haben, sehr ähnlich sein. Er sollte sich nur durch ein paar Zahlen, die aber für unser Programm nicht wichtig sind, von dem ersten unterscheiden. Er sollte also in etwa folgendes Aussehen haben:

```
***Illegal Instruction at $C44454
D0=00000005 00000012 00000000 00000000 00000000 00000000 00000000 00000000
A0=00000000 00000000 00000000 00000000 00000000 00000000 00000000 00C39BFC
SSP=00C3AB9C USP=00C39BFC SR=8000 PC=C44454 ILLEGAL
```

Damit haben wir also dem Prozessor bei der Abarbeitung unseres ersten Programms zugeschaut.

Hinweis: Ein Programm Schritt für Schritt abzuarbeiten, ist oft sehr nützlich. Zum einen können Sie sich fehlerhafte Programme Schritt für Schritt zeigen lassen und somit herausfinden, an welcher Stelle des Programms der Fehler liegt. Zum anderen können Sie sich die Funktionsweise neuer Befehle in Ruhe anschauen und prüfen, ob sie wirklich das tun, was Sie erwartet haben.

Wir wollen noch einmal kurz zusammenfassen, wie wir die Addition in einem Maschinensprache-Programm verwirklicht haben.

Zusammenfassung

Schauen wir uns den Ablauf des Programms noch einmal in einer kurzen Zusammenfassung an:

- Das Label START: ermöglicht es uns, herauszufinden, an welcher Stelle das Programm abgelegt worden ist.

- Das NOP ermöglicht es uns, dem Prozessor von Anfang an bei der Arbeit zuzuschauen, da dieser Befehl die Wirkung hat, daß zunächst einmal "keine Operation" ausgeführt wird. Verwendet man nämlich den Befehl "S" für Step, Einzelschritt, wird erst der zweite Befehl angezeigt.

- Mit MOVE erreichen wir, daß Werte übertragen werden, in unserem Fall in Register. Dabei ist darauf zu achten, ob es sich um ein Langwort, Wort oder Byte handelt. Dem MOVE sind dann die entsprechenden Anhängsel durch einen Punkt getrennt anzuhängen.

  L Langwort
  W Wort
  B Byte

- Soll nur ein Wort übertragen werden, kann das Anhängen von "W" auch ausbleiben, da der Befehl MOVE ohne Anhängsel automatisch ein Wort überträgt.

- Der Befehl ADD führt die Addition aus. Das Ergebnis dieser Addition steht immer in dem Register, welches hinter dem Komma angegeben ist. Auch hierbei ist darauf zu achten, ob es sich bei der Addition um Langworte, Worte oder Bytes handelt.

- Das ILLEGAL bricht das Programm ab und übergibt die Kontrolle wieder dem SEKA. Deshalb sollten alle unsere Programme erst einmal mit dem Befehl ILLEGAL enden.

3. Amiga ein - LED aus

Kommen wir nun zu einem weiteren Beispiel. Wir wollen ein kleines Programm schreiben, welches direkt einen sichtbaren Eingriff auf unseren Computer ausübt. Und zwar wollen wir die LED des Computers ausschalten, obwohl Ihr Computer weiter betriebsbereit ist. Wie ist dieses nun zu bewerkstelligen?

3.1 Wo liegt die LED im Amiga

Nun, dazu muß man zunächst einmal wissen, an welcher Speicherstelle sich die Information darüber befindet, ob die LED eingeschaltet ist oder nicht.

Diese Information liegt in der Speicherstelle $BFE001 (hexadezimale Schreibweise). Von dieser Speicherstelle ist das zweite Bit zuständig für die LED. Ist dieses Bit auf 1 gesetzt, so ist die LED aus, ist es auf 0 gesetzt, leuchtet sie. Dazu muß man noch wissen, daß die Bits im Amiga immer von 0 an gezählt werden. Das erste Bit hat die Nummer 0, das zweite Bit die Nummer 1 und so fort. Verantwortlich für den Zustand der LED ist also das zweite Bit mit der Nummer 1.

Wichtig ist es auch, zu wissen, daß jedes Bit dieser Speicherstelle seine ganz spezielle Bedeutung hat. Das ist auch der Grund dafür, das man nicht so einfach alle Bits beliebig setzen kann. Wir werden das zwar im folgenden kleinen Programm tun, aber wir wissen auch, daß in diesem konkreten Beispiel nichts passieren kann und müßten für eine absolut korrekte Vorgehensweise spezielle Befehle verwenden, mit denen wir uns lieber erst etwas später beschäftigen.

Wie löscht man Zeilen im Editor

Bevor wir unser nächstes Programm eingeben können, müssen wir natürlich das bisherige Programm loswerden. Wir könnten

nun Zeichen für Zeichen mit <Del> oder <Backspace> löschen, aber dafür gibt es einen speziellen Befehl Z (für Zap = Löschen). Hinter Z gibt man die Anzahl der Zeilen an, die ab der aktuellen Zeile im Editor gelöscht werden sollen.

In unserem Fall aktivieren wir mit <Esc> den Editor, bewegen den Cursor in die erste Zeile und geben ein:

```
Z6
```

Anschließend zeigt der SEKA uns alle gelöschten Zeilen im Befehlsbereich an und meldet:

```
1 ** End of File
```

3.2 Die LED wird ausgeschaltet

Nun können wir unser zweites Programm eingeben. Aktivieren Sie mit <Esc> den Editor, und geben Sie dort folgendes ein:

```
START:
MOVE.B #02,$BFE001
ILLEGAL
```

Aktivieren Sie mit <ESC> wieder den Befehls-Modus, und assemblieren Sie dies Programm. Geben Sie also ein "A" ein. Die Frage nach den Optionen können Sie wieder getrost übergehen.

Sind keine Fehler in Ihrem Programm aufgetreten, erhalten Sie wie üblich die Meldung:

```
NO ERRORS
```

Starten Sie nun dieses Programm, in dem Sie ein "G" eingeben und die Frage nach den Breakpoints mit <Return> übergehen. Es erscheint die obligatorische Registeranzeige. Doch was ist nun passiert, Sie können nichts feststellen?

Nun, dann schauen Sie sich doch einmal die LED Ihres Amigas an. Sie ist ausgeschaltet, obwohl Ihr Computer ganz normal funktioniert.

Wir speichern ein Programm

Bevor wir zu einer Erklärung kommen, wollen wir dieses "schöne" Programm einmal abspeichern.

Um ein Programm zu speichern, müssen wir uns im Befehls-Modus befinden. Dort geben Sie zunächst einmal ein "W" für "Write" (Schreiben) ein und drücken die Eingabe-Taste. Sie werden nun nach dem "FILENAME", dem Dateinamen, gefragt. Wir haben hier als Dateinamen "LED_AUS" gewählt. Sie sollten gegebenenfalls den kompletten Pfadnamen, also auch das Laufwerk, angeben. Die Eingabe könnte also folgendermaßen aussehen:

```
DF0:LED_AUS
```

Nach Bestätigung mit der Eingabe-Taste wird Ihr Programm unter dem Namen, den Sie angegeben haben, abgespeichert. Der SEKA hängt beim Speichern an den Dateinamen automatisch ein ".S" an. Daran erkennt er, das es sich um eine SOURCE-Datei handelt. SOURCE-Dateien sind Quellen für das später auszuführende Programm.

Laden

Wollen Sie ein einmal gespeichertes Programm wieder in den Editor laden, müssen Sie das, was vorher im Editor stand, erst einmal löschen. Dieses geschieht, in dem Sie im Befehls-Modus eingeben:

```
KS
```

Diese Abkürzung steht für "Kill Source" und hat somit die Bedeutung, daß die gesamte Quelldatei im Amiga gelöscht wird.

Danach erfolgt eine Sicherheitsabfrage, die Sie mit "Y" für "Ja" beantworten. Aktivieren Sie mit <Esc> den Editor, und Sie werden sehen, daß alles bis auf das <End> gelöscht wurde. Nun können Sie ihr Programm in den Editor laden, in dem Sie im Befehls-Modus ein "R" für "Read" (Lesen) eingeben. Danach werden Sie nach dem "FILENAME", also dem Dateinamen gefragt. Geben Sie am besten den gesamten Pfad des zu ladenden Programms ein. In unserem Beispiel wäre das:

```
DF0:LED_AUS
```

Nach Bestätigung mit der Eingabe-Taste befindet sich das Programm im Editor.

Die Erklärung des Programms

Doch kommen wir noch einmal zu dem Programm, welches die LED ausschaltet. Wie schon gesagt, ist für die LED das zweite Bit der Speicherstelle $BFE001 zuständig. Um die LED auszuschalten, müssen wir dieses Bit ändern, d.h. auf 1 setzen. Dieses können wir erreichen, in dem wir das gesamte Byte dieser Speicherstelle ändern. Wir müssen also an den Befehl MOVE ein ".B" anhängen. Nun haben wir Ihnen jedoch vorhin gesagt, daß wir nicht einfach jedes Bit dieser Speicherstelle beliebig ändern dürfen, da jedes seine ganz spezielle Bedeutung hat, und es daher bei beliebigem Ändern dieser Bits zu unerwünschten Reaktionen kommen kann. Mit unserem Befehl:

```
MOVE.B #02,$BFE001
```

jedoch setzen wir einfach alle Bits bis auf das zweite (mit der Nummer 1) auf Null. Denn wir schreiben in die Speicherstellen $BFE001 den Wert 2, und der sieht binär folgendermaßen aus:

Bitnr.:	7	6	5	4	3	2	1	0
Zustand:	0	0	0	0	0	0	1	0

Wir dürfen dieses in diesem Fall ausnahmsweise tun, da wir wissen, daß es nicht zu unerwünschten Operationen dabei kommt. Aber im allgemeinen sollte man bei solchen Dingen äu-

ßerste Vorsicht an den Tag legen. Wir werden Ihnen später auch noch eine Möglichkeit zeigen, wie Sie direkt ein einzelnes Bit einer Speicherstelle ändern können, ohne daß die anderen Bits davon betroffen sind.

LED wieder einschalten

Doch nun wollen wir die LED wieder einschalten, das zweite Bit der Speicherstelle $BFE001 also wieder auf Null setzen. Am besten ändern wir das bestehende Programm einfach ab. Dazu aktivieren Sie mit <Esc> den Editor, in dem hoffentlich Ihr Programm zum Ausschalten der LED noch steht. Ansonsten laden Sie es, wie vorher beschrieben, wieder in den Editor. Im Editor ändern Sie einfach die Zeile:

```
MOVE.B #02,$BFE001
```

in:

```
MOVE.B #00,$BFE001
```

Ihr gesamtes Programm müßte also folgendermaßen aussehen:

```
START:
MOVE.B #00,$BFE001
ILLEGAL
```

Aktivieren Sie den Befehls-Modus, assemblieren Sie das Programm, und starten Sie es. Schauen Sie sich Ihre LED an, und Sie leuchtet wieder. Kein schlechter Erfolg oder?

3.3 Zahlensysteme und Adressierungen

Im folgenden Abschnitt wollen wir Ihnen an diesem Programm verschiedene Zahlenangaben und Adressierungen vorstellen.

Verschiedene Zahlensysteme

Vielleicht ist Ihnen schon aufgefallen, daß wir bei unserem Programm, welches die LED ausschaltet, diesmal statt des Wertes "#$02" einfach den Wert "#02", also den dezimalen Wert, geschrieben haben. Nun, der Grund dafür ist ganz einfach. Diese beiden Werte sind identisch. Denn das hexadezimale "#$02" entspricht ja dem dezimalen "#02". Wir hätten an dieser Stelle auch "#%10" schreiben können, denn dieses ist die duale Schreibweise der Dezimalzahl "2". Daß wir sowohl bei hexadezimaler als auch bei dezimaler Schreibweise die "02" geschrieben haben, hat wieder den einfachen Grund, daß es üblich ist, immer ein ganzes Byte zu schreiben.

Die Umwandlung der Zahlen mit dem SEKA

Sie haben beim SEKA die Möglichkeit, sich die verschiedenen Zahlen in hexadezimaler und dezimaler Schreibweise anzuschauen. Dazu geben Sie im Befehls-Modus ein "?" ein und dahinter die Zahl, dessen hexadezimalen und auch dezimalen Wert Sie wissen möchten. Wollen Sie Hexzahlen umwandeln, müssen Sie vor dem Wert das "$"-Zeichen eingeben, bei Dualzahlen das "%"-Zeichen und bei Dezimalzahlen einfach nur den Wert. Nach Drücken der Eingabe-Taste gibt Ihnen der SEKA die Werte nun als Hexzahl und als Dezimalzahl aus. Dieses ist manchmal sehr günstig, da wir daran gewöhnt sind, immer mit Dezimalzahlen zu rechnen, und es uns daher oft schwer fällt, Zahlen in Hexzahlen umzurechnen.

Als nächstes wollen wir uns ein wenig mit verschiedenen Adressierungsmöglichkeiten beschäftigen, also mit verschiedenen Möglichkeiten, Quellen und Ziele zu benennen.

Adressierungsmöglichkeiten

Es gibt verschiedene Möglichkeiten der Adressierung. Um diese zu erklären, wollen wir uns so eine Adressierung noch einmal anschauen. Eine Adressierung besteht aus einer Quelle und einem Ziel. In unserem Fall bei dem Programm zum Ausschalten der LED ist der Operand "#02" die Quelle und der Operand

"$BFE001" das Ziel. Wodurch unterscheiden sich diese beiden Operanden nun? Vereinfacht gesagt: Die Quelle ist in diesem Fall ein Wert, also eine feste Zahl, während das Ziel eine Speicherstelle ist. Damit haben wir auch schon unsere ersten beiden Adressierung kennengelernt, die unmittelbare und die absolute Adressierung.

Unmittelbare Adressierung

Die Adressierung der Quelle nennt man in unserem Fall unmittelbare Adressierung. Merken kann man sich das so, daß die Quelle unmittelbar als Zahl angegeben wird. Um eine unmittelbare Adressierung von der absoluten Adressierung zu unterscheiden, wird der Zahl das Zeichen "#" vorangestellt.

Absolute Adressierung

Die Adressierung des Ziels ist in dem genannten Beispiel eine absolute Adressierung, da es sich nicht um einen Wert handelt, sondern um die Adresse einer Speicherstelle, die direkt angesprochen wird. Man nennt diese Art der Adressierung auch direkte Adressierung. Absolut oder direkt heißt diese Adressierung auch, weil die Adresse nicht relativ zu einer bestimmten Speicherstelle oder relativ zu einem Register angegeben wird.

Ziele können nie unmittelbar adressiert werden, denn wie will man in eine Zahl hineinschreiben. Also ist eine Befehlszeile:

```
MOVE.B $BFE001,#00
```

nicht möglich und wird beim Assemblieren auch vom SEKA mit der Meldung "Illegal Operand" beanstandet.

Quellen hingegen können nicht nur unmittelbar, sondern auch absolut adressiert werden. Dieses wollen wir mit einem Beispiel zeigen.

```
MOVE.L $BFE001,D0
```

Wir haben also den Inhalt der Speicherstelle $BFE001 ins Datenregister D0 übertragen. Hier handelt es sich somit um eine absolute Adressierung für die Quelle.

Indirekte Adressierung

Eine weitere Art der Adressierung ist die indirekte Adressierung. Auch diese wollen wir anhand eines Beispiels erklären:

```
MOVE.B #$02,(A0)
```

Mit diesem Befehl wird der Inhalt der Speicherstelle geändert, deren Adresse in A0 steht. Wenn A0 beispielsweise die Adresse der LED enthält, wird somit wieder die LED geändert. Mit diesen Adressierungsmöglichkeiten können wir ein neues kurzes Programm zum Ausschalten der LED schreiben.

Dazu geben Sie im Editor folgendes Programm ein:

```
START:
MOVE.L #$BFE001,A0
MOVE.B #$02,(A0)
ILLEGAL
```

Was haben wir mit diesem Programm gemacht? Wie haben die gewünschte Adresse der LED in das Register A0 geschrieben und anschließend die Zahl 2 in die in A0 angegebene Adresse. Das Besondere der indirekten Adressierung sind immer die runden Klammern um das Register, in diesem Fall also um A0.

Ein Unterschied zwischen Adreß- und Datenregistern

Interessant ist in diesem Zusammenhang auch der Unterschied zwischen Daten- und Adreßregistern. Sollten Sie vielleicht einmal versuchen, in diesem Programm statt des verwendeten Adreßregisters A0 das Datenregister D0 zu verwenden, so werden Sie beim Assemblieren feststellen, daß dieses nicht zulässig ist. Sie erhalten eine Fehlermeldung.

```
**Illegal Operand
   3 MOVE.B #$02,(D0)
```

Es scheint in diesem Fall nicht erlaubt zu sein, Datenregister zu verwenden. Doch warum nicht?

Nun, hier zeigt sich, daß Datenregister nur Daten verwalten können. Da es sich in unserem Fall jedoch um eine Adresse handelt, die verwaltet werden muß, können wir kein Datenregister zur Verwaltung benutzen. Adreßregister hingegen können sowohl Adressen als auch Daten verwalten. Allerdings können nur mit den Datenregistern die vielen Rechenbefehle verwendet werden.

Es gibt natürlich noch weitere Unterschiede zwischen Adreß- und Datenregistern. Auf die wollen wir jedoch erst später zurückkommen.

Langworte

Wir wollen in diesem Zusammenhang noch auf etwas anderes aufmerksam machen, und zwar auf eine Besonderheit von Langworten. Wenn wir versuchen in der Zeile:

```
MOVE.B #$02,(A0)
```

statt des Bytes ein Langwort zu übertragen (MOVE.L), so hätten Sie erneut eine Fehlermeldung erhalten, diesmal allerdings erst nach (!!!) dem Starten des Programms. Dieses liegt daran, daß Langworte nur an eine gerade Adresse übertragen werden dürfen. Da es sich bei BFE001 jedoch um eine ungerade Adresse handelt, kommt es bei der Abarbeitung des Programms zu einem Fehler.

3.4 Namen statt Zahlen

Wir wollen nun ein paar Vorbereitungen für unser nächstes Programm treffen, wir wollen Labels vergeben. Hier stellt sich die Frage, was sind Labels überhaupt und wozu sind sie notwendig bzw. vorteilhaft?

In unserem Fall ist ein Label erst einmal ein Name für eine Konstante. Haben Sie beispielsweise ein Programm, in dem eine bestimmte Zahl des öfteren vorkommt, so ist es günstig, dieser Konstanten einen Namen zu geben, statt mühselig immer wieder den Wert der Konstanten einzugeben.

Welche Namen sind sinnvoll?

Am günstigsten ist es einer Konstanten einen Namen zu geben, der etwas über diese Konstante aussagt. Nehmen wir als Beispiel einfach die Adresse der LED. Wir wollen als nächstes ein Programm schreiben, in dem diese Adresse des öfteren angesprochen werden soll. Geben wir dieser Adresse doch den Namen LED. Um einer Konstanten einen Namen zu geben, (wir werden in diesem Zusammenhang immer von einem Label sprechen, weil für einen Assembler zwischen einer Konstanten und einer Sprungmarke kein Unterschied besteht), müssen Sie vor dem Start eines Programms den Namen des Labels angeben, gefolgt von einem Gleichheitszeichen und der Konstanten, die Sie als Label setzen wollen. In unserem Fall würde eine solche Zeile folgendermaßen aussehen:

```
LED = $BFE001
```

Viele Änderungen auf einen Streich

Ein weiterer großer Vorteil eines Labels ist, daß, falls Sie einen falschen Wert für etwas eingesetzt haben und Sie dieses erst nachher merken, es ein leichtes ist, diesen Wert zu ändern. Anstatt den Wert nun überall heraussuchen zu müssen, können Sie einfach die Zeile ändern, in der Sie den Wert für das Label festgesetzt haben.

Eine solche Änderung ist nicht nur sehr komfortabel, sondern schließt auch mögliche Fehlerquellen aus. Wie schnell vergißt man nämlich bei der Änderung in etwas längeren Programmen an einigen Stellen die Änderung, und anschließend funktioniert das Programm natürlich nicht korrekt. Wird die Zahl nur einmal zu Beginn des Programms festgelegt, kann das nicht passieren.

3.5 Wir lassen die LED blinken

Damit haben wir einige Möglichkeiten kennengelernt, mit denen wir unser Programm erweitern und verbessern können. Mit diesem Programm wollen wir unsere LED zum Blinken bringen. Dazu wollen wir zunächst ein Programm erstellen, das unsere LED ein- und wieder ausschaltet.

Sie werden sicher sagen, daß dieses kein großes Problem sein kann. Denn schließlich haben Sie ja sowohl ein Programm zum Ausschalten der LED als auch eines zum Einschalten geschrieben. Und wenn Sie nun beide Programme hintereinander abarbeiten lassen, wird sich die LED erst aus- und dann einschalten. Schauen wir uns doch einmal an, wie so ein Programm aussehen würde:

```
START:
MOVE.B #02,$BFE001
ILLEGAL
START:
MOVE.B #00,$BFE001
ILLEGAL
```

Nun, Sie werden sicher schon selbst merken, daß dieses Programm nicht funktionieren kann. Denn nach dem Ausschalten der LED stößt der Prozessor auf ein ILLEGAL, bricht ab und übergibt die Kontrolle dem SEKA. Das Ende des Programms, also das Einschalten der LED kann somit überhaupt nicht mehr geschehen. Damit muß also das ILLEGAL zwischen den beiden Programmen weggelassen werden. Also könnten Sie das ILLEGAL löschen, dann könnten Sie es assemblieren, und schon hätten wir die nächste Fehlermeldung:

```
**Double Symbol
   3 START:
```

Nun könnten Sie also auch noch diese Zeile löschen. Ihr Programm würde somit folgendermaßen aussehen:

```
START:
MOVE.B #02,$BFE001
MOVE.B #00,$BFE001
ILLEGAL
```

Wenn Sie nun dieses Programm assemblieren und schließlich abarbeiten lassen, so wird das auch funktionieren, was Sie daran erkennen können, daß der Block erscheint, der Sie darauf hinweist, daß der Prozessor auf ein ILLEGAL gestoßen ist. Doch wenn Sie während des Ablaufs des Programms Ihre LED beobachtet haben, werden Sie sagen, daß es überhaupt nicht funktioniert hat.

Daß es funktioniert hat, obwohl Sie nichts gesehen haben, können wir Ihnen beweisen. Dafür assemblieren Sie das Programm neu und lassen es nun Schritt für Schritt abarbeiten. Wenn Sie nun auf die LED achten, werden Sie feststellen, daß sie tatsächlich aus und an geht. Warum haben Sie dann beim Abarbeiten des Programms nichts gesehen? Dies liegt ganz einfach daran, daß das Programm so schnell abgearbeitet wird, daß das Aus- und Einschalten der LED nicht wahrgenommen werden kann.

Das ist aber etwas, was wir nicht wollen. Wie können wir dieses nun verhindern?

Wir wollen in unser Programm eine Warteschleife einbauen. Diese Warteschleife bewirkt, daß zwischen dem Aus- und Einschalten der LED Zeit vergeht, und wir somit sehen können, wie das Programm die LED aus- bzw. einschaltet. Die Warteschleife ist ganz einfach aufgebaut. Zuerst wird einem Datenregister eine recht hohe Zahl zugewiesen, und dann wird von diesem Wert immer eins abgezogen, bis der Wert Null ist. Erst jetzt wird die LED wieder eingeschaltet.

Um so ein, schon etwas aufwendigeres, Programm zu schreiben, ist es notwendig, unser bisheriges Programm neu zu strukturieren. Wir könnten es zwar auch so lassen, wie es im Moment ist, würden dann aber irgendwann die Übersicht verlieren.

Als ersten Schritt können wir erst einmal, wie oben schon erklärt, Konstanten einen Namen geben. In unserem Fall haben wir nur eine einzige Konstante und zwar die Konstante, die die Adresse der LED enthält. Geben wir ihr also auch diesen Namen. Unser oben beschriebenes Programm würde dann folgendermaßen aussehen:

```
START:
LED = $BFE001
MOVE.B #02,LED
MOVE.B #00,LED
ILLEGAL
```

Unser Programm hat damit zwar eine Zeile mehr, wird aber auch übersichtlicher und verständlicher.

Nun wollen wir uns einmal überlegen, wie unser Programm mit der Warteschleife aufgebaut sein muß, was es also nacheinander ausführen muß.

1.... LED ausschalten

2.... Warteschleife

3.... LED einschalten

Genau so soll unser Programm auch strukturiert sein. Wir wollen also ein Hauptprogramm schreiben, in dem genau dieser Ablauf beschrieben wird. Aus diesem Hauptprogramm heraus wollen wir dann die entsprechenden Unterprogramme aufrufen. Auch den Unterprogrammen kann man wieder Namen, also Labels geben. In diesem Fall haben die Namen jedoch eine andere Bedeutung. Es sind nämlich sogenannte Sprungadressen.

3.6 Wir erstellen ein Unterprogramm

Um mit Unterprogrammen zu arbeiten, müssen wir zunächst einige neue Befehle kennenlernen. Zum einen einen Befehl, mit

dem wir ein Unterprogramm, auch Unterroutine genannt, aufrufen können, zum anderen einen Befehl, um wieder in das Hauptprogramm zurückzukehren.

Der Befehl JSR

Der Befehl, mit dem wir ein Unterprogramm aufrufen können, lautet:

```
JSR
```

Dies ist die Abkürzung für Jump Subroutine, was übersetzt bedeutet: Springe zur Unterroutine. Dieser Hinweis soll nur als kleine Gedächnisstütze dienen. Wir wollen Ihnen noch eine weitere mitgeben. Vielleicht kennen Sie Unterroutinen ja schon von der Programmiersprache BASIC her. Dort ist dieser Befehl vergleichbar mit dem Befehl GOSUB.

Diesem Befehl muß immer der Name einer Sprungadresse, also ein Label, nachgestellt werden. Das Label muß also vor dem eigentlichen Unterprogramm stehen, und des weiteren muß es mit einem Doppelpunkt abgeschlossen sein. Dieses ist sehr wichtig, damit der SEKA erkennen kann, daß es sich hierbei um ein Label handelt. Der SEKA trägt beim Assemblieren die richtige Adresse für das Label ein, damit der Prozessor weiß, wo er hinspringen, bzw. wo er mit der Abarbeitung des Programms fortfahren soll.

Trifft der Prozessor also auf einen JSR-Befehl, sucht er im Programm nach der entsprechenden Adresse für das Label und fährt erst dort mit der Abarbeitung des Programms fort. Der Befehl JSR erfordert am Ende der Unterroutine immer ein RTS.

Hinweis: Sie können ein Programm zwar auch in einem Unterprogramm mit dem Befehl ILLEGAL beenden, aber dies geht wirklich nur so lange gut, wie Sie das Programm im SEKA abarbeiten lassen. Wir wollen aber im Verlauf des Buches auch Programme erstel-

len und speichern, die jederzeit ohne den SEKA genutzt werden können. Deshalb: Unterprogramme immer mit RTS beenden.

Der Befehl RTS

Um nun aus einer Unterroutine wieder ins Hauptprogramm zurückzukehren, brauchen wir den neuen Befehl.

```
RTS
```

Dies ist die Abkürzung für RETURN SUBROUTINE, was soviel bedeutet wie: Kehre aus der Unterroutine zurück. Wenn Sie schon mit BASIC programmiert haben, können Sie den Befehl mit dem Befehl RETURN vergleichen.

Trifft der Prozessor auf den Befehl RTS, so springt er wieder dorthin zurück, wo er vor dem Befehl JSR hergekommen ist, und fährt dort mit der Abarbeitung des Programms fort. Die Rücksprungadressen befinden sich in einem Speicherbereich, dem sogenannten Stack. Dabei zeigt das Stackregister (engl. Stackpointer SP) immer auf die nächste passende Rücksprungadresse.

Hinweis: Der Prozessor 68000 benutzt als Stackregister immer das Adreßregister A7. Wenn Sie also während unserer Beispiele einmal etwas ausprobieren und Veränderungen vornehmen wollen, dürfen Sie keinesfalls einfach dieses Register verwenden, weil der Prozessor sonst nicht mehr die Rücksprungadresse finden kann.

3.7 Wir programmieren eine Warteschleife

Eines unserer Unterprogramme, die wir aufrufen wollen, sollte unsere Warteschleife sein. Wie schon vorher angesprochen, wollten wir unsere Warteschleife so aufbauen, daß der Prozessor von

einer sehr hohen Zahl so lange eins abzieht, bis das Ergebnis gleich null ist. Auch hierfür müssen wir neue Befehle kennenlernen.

Der Befehl SUB

Als erstes müssen wir einen Befehl haben, der es uns ermöglicht, zu subtrahieren. Dies ist der Befehl:

```
SUB
```

Dieser Befehl funktioniert ähnlich wie der Befehl ADD. Es muß mitgeteilt werden, ob es sich bei dem zu subtrahierenden Wert um ein Byte, ein Wort oder ein Langwort handelt. Des weiteren wird auch bei der Subtraktion das Ergebnis in dem Register, welches hinter dem Komma steht, abgelegt. Dieses soll vorerst zu diesem Befehl genügen.

Der Befehl BNE

Um nun eine Schleife zu konstruieren, müssen wir dem Prozessor mitteilen, daß er das Ergebnis unserer Subtraktion überprüfen soll. So lange das Ergebnis ungleich Null ist, soll er wieder eins abziehen und erneut überprüfen.

Für den Vergleich, ob das Ergebnis der Subtraktion gleich Null ist, brauchen wir einen neuen Befehl. Dieser neue Befehl ist der Befehl BNE. Er muß wiederum von einem Label gefolgt werden, zu dem der Prozessor zurückkehrt, falls das Ergebnis der Subtraktion noch ungleich Null ist. Was macht nun dieser Befehl?

Dafür müssen wir ein wenig ausholen. Der Befehl BNE überprüft den Zustand des Zeroflags. Dieses Zeroflag ist in einem speziellen Register des 68000, dem Statusregister, vorhanden. Eigentlich ist das Zeroflag nur ein Bit in diesem Register. Man nennt dieses und - auch andere Bits - Flags, weil sie wie eine Flagge ein Signal geben.

Die Flags im Statusregister - und damit auch das Zeroflag - werden vom Prozessor völlig selbständig verändert. Das Zeroflag ist gleich Null, so lange das Ergebnis einer Operation ungleich Null ist. Sobald bei einer Operation das Ergebnis Null erreicht wird, wird das Zeroflag auf eins gesetzt. Der Befehl BNE ist nun die Abkürzung von BRANCH NOT EQUAL, was übersetzt soviel bedeutet wie: Verzweige so lange nicht gleich. Er wird meist nach einer Subtraktion oder einem Vergleich verwendet.

Trifft der Prozessor nun auf diesen Befehl, wird überprüft, ob das Zeroflag gesetzt, also gleich eins ist. Ist dies nicht der Fall, wird zu dem Label gesprungen, welches hinter dem Befehl angegeben ist. Ist das Zeroflag gesetzt, also gleich eins und somit auch das Ergebnis der Operation gleich Null, wird nicht wieder zum Label gesprungen, sondern mit dem nächsten Befehl, der hinter dem BNE steht, fortgefahren.

Damit haben wir alle Befehle, die wir für unser Programm brauchen, erklärt. Wir wollen zunächst einmal nur das Programm für die Schleife beschreiben:

```
PAUSE:
WARTE = 100000
MOVE.L #WARTE,D0
SCHLEIFE:
SUB.L #01,D0
BNE SCHLEIFE
ILLEGAL
```

Damit haben wir im Grunde alles, was wir für unser Programm benötigen. Erinnern Sie sich noch an die Struktur, die unser Programm haben sollte:

1.... LED ausschalten

2.... Warteschleife

3.... LED einschalten

Für alle drei Punkte kennen wir nun die Programme. Jetzt ist nur noch die Frage offen, auf welche Art und Weise wir diese Programme am besten zusammenfügen. Dazu einige Tips:

- Alle Variablen eines Programms sollten am Anfang des Programms definiert werden.

- Um eine gute Übersicht über Ihr Programm zu haben, sollten Sie Haupt- und Unterprogramme jeweils durch eine Leerzeile trennen.

- Zu Ihrer Orientierung sollten Sie die Haupt- und Unterprogramme mit Überschriften versehen. Dieses geschieht, in dem Sie der jeweiligen Überschrift ein Semikolon voranstellen. Das heißt für den Assembler, daß das nun Folgende kein Befehl ist. Das Semikolon hat somit die gleiche Bedeutung wie ein REM in der Programmiersprache BASIC.

- Schleifen sollten Sie etwas einrücken, also einfach Leerzeichen davor lassen, damit Sie sofort ins Auge fallen.

Kommen wir nun zu unserem ersten etwas umfangreicheren Programm. Dieses ist nun auch schon schön durchstrukturiert und somit haben wir eine gute Übersicht. Auch die Befehle werden deutlicher, wenn wir uns das entsprechende Programm einmal in seiner endgültigen Form anschauen.

```
START:

; ***** Variablen definieren *****
LED   =$BFE001
WARTE = 100000

; ***** Hauptprogramm *****
JSR AUS
JSR PAUSE
JSR EIN
ILLEGAL

; ***** Unterprogramm LED ausschalten *****
AUS:
MOVE.B #02,LED
RTS

; ***** Unterprogramm Warteschleife *****
PAUSE:
MOVE.L #WARTE,D0
SCHLEIFE:
   SUB.L #01,D0
```

```
    BNE SCHLEIFE
RTS

; ***** Unterprogramm LED einschalten *****
EIN:
MOVE.B #00,LED
RTS
```

Nachdem Sie dieses Programm in den Editor eingegeben haben, gehen Sie zurück in den Befehls-Modus. Dort müssen Sie das Programm assemblieren. Haben Sie bei der Eingabe des Programms keine Fehler gemacht, werden Sie als Meldung erhalten:

```
NO ERRORS
```

Sollten Sie jedoch eine Fehlermeldung erhalten, vergleichen Sie das von Ihnen eingegebene Programm noch einmal mit dem abgedruckten Programm, und beheben Sie die Fehler. Vergessen Sie nicht, nach einer Korrektur neu zu assemblieren.

Ja, und nun kann es losgehen. Lassen Sie Ihr Programm doch einfach einmal abarbeiten. Geben Sie ein:

```
G
```

Sobald Sie die Eingabe-Taste gedrückt haben, können Sie beobachten, wie zuerst die LED ausgeschaltet und nach einer Weile wieder eingeschaltet wird.

Sie können sich natürlich auch dieses Programm Schritt für Schritt vorführen lassen. Doch dazu ein kleiner Tip: Wie Sie sich sicherlich noch erinnern, haben wir in dem Programm eine Warteschleife eingebaut, um die LED auch richtig blinken zu sehen. Den Zähler für diese Warteschleife haben wir auf 100000 gesetzt. Somit wird diese Schleife 100000mal durchlaufen. Wir wissen zwar nicht, ob es Ihnen Spaß macht, sich diese ganzen Einzelschritte anzuschauen, aber wir vermuten einmal, daß das nicht der Fall ist.

Um nun aber trotzdem den Ablauf des Programms verfolgen zu können, setzen Sie den Zähler einfach auf einen kleineren Wert.

Da wir den Zähler als eine Variable ganz am Anfang unseres Programms gestellt haben, müssen Sie nur die Zeile:

```
WARTE = 100000
```

umwandeln in:

```
WARTE = 3
```

So, jetzt können Sie sich das Programm Schritt für Schritt abarbeiten lassen. Sie erinnern sich, nach dem Assemblieren geben Sie im Befehls-Modus ein:

```
s
```

Wenn Sie jetzt Step by Step Ihr Programm abarbeiten lassen, wollen wir Ihnen den Ablauf des Programms auch noch einmal erklären.

Als erstes legen wir das Anfangslabel für unser Programm fest. Wir haben es erneut "START:" genannt. Danach legen wir die für das Programm benötigten Variablen fest. Da ist einmal die Variable LED, die die Speicherstelle angibt, in der sich die Informationen über die LED befinden, und zum anderen der Startwert für unseren Zähler, den wir für die Schleife benötigen.

Als nächstes kommt unser Hauptprogramm, in dem wir alle Unterprogramme aufrufen, die wir für unser gesamtes Programm benötigen. Die Struktur des Hauptprogramms ist im großen und ganzen identisch mit unseren Überlegungen zum Aufbau des Programms. Als erstes LED ausschalten, dann die Warteschleife und als letztes die LED wieder einschalten.

Der erste Befehl, auf den unser Programm trifft, ist:

```
JSR AUS
```

Dies ist der Sprungbefehl in die erste Unterroutine. Der Prozessor springt also zur Adresse des Labels der Unterroutine und

setzt dort seine Arbeit fort. In dieser Unterroutine wird die LED ausgeschaltet. Danach trifft der Prozessor auf den Befehl:

```
RTS
```

Dieser Befehl sagt ihm, daß er wieder ins Hauptprogramm zurückkehren soll und zwar direkt hinter den Befehl der ihn zum Verlassen des Hauptprogramms bewegt hat, also hinter den Befehl:

```
JSR AUS
```

Hier trifft er sofort auf einen erneuten Sprungbefehl, der Sprungbefehl in die Unterroutine, die unsere Warteschleife enthält.

Innerhalb dieser Warteschleife wird zunächst einmal der Startwert für den Zähler in ein Datenregister übertragen. Danach beginnt die eigentliche Schleife. Innerhalb dieser Schleife wird vom Datenregister der Wert "1" abgezogen, und danach wird überprüft, ob der Inhalt des Registers, in dem sich der Zähler der Warteschleife befindet, gleich Null ist.

Dieses geschieht nicht direkt, sondern über den Umweg des Zeroflags. Der Befehl BNE überprüft nicht den Inhalt des Datenregisters, sondern überprüft, ob das Zeroflag gesetzt, also gleich "1" ist und damit das Ergebnis unserer Operation gleich Null ist oder nicht. So lange das Zeroflag nicht gesetzt ist, kehrt der Prozessor zu dem Label zurück, welches hinter dem Befehl angegeben ist. Ist das Zeroflag dann gesetzt, arbeitet der Prozessor den nächsten Befehl ab, in unserem Fall ein RTS, er kehrt also wieder zum nächsten Befehl ins Hauptprogramm zurück.

Dies ist wieder ein Sprungbefehl, der letzte, der die LED wieder einschaltet. Durch das anschließende RTS kehrt der Prozessor wieder ins Hauptprogramm zurück und trifft dort auf ein ILLEGAL, was ihn zum Abbruch des Programms zwingt.

Sie werden sich nun sicher fragen, warum dieser letzte Sprung ins Hauptprogramm noch nötig war, wir hätten das ILLEGAL doch genausogut an das Ende der letzten Unterroutine setzen können. Das stimmt zwar, aber Sie erinnern sich vielleicht bei der Erläuterung von Unterprogrammen an unseren Hinweis, daß ein Abbruch des Programms in einem Unterprogramm nur um SEKA funktioniert, nicht aber, wenn das Programm später einmal von DOS aus gestartet werden soll.

3.8 Zusammenfassung

Wir wollen noch einmal kurz zusammenfassen, was wir in diesem Kapitel alles kennengelernt haben.

Da sind zunächst einmal vier neue Befehle:

- **JSR:** Dieser Befehl ruft eine Unterroutine auf. Hinter dem Befehl muß stets das Label der Unterroutine, die aufgerufen werden soll, stehen.

- **RTS:** Mit diesem Befehl kehrt man aus einer Unterroutine direkt hinter den Befehl zurück, von wo aus man in die Unterroutine gesprungen ist.

- **SUB:** Dieser Befehl funktioniert ähnlich wie der Befehl ADD. Er wird zum Subtrahieren benötigt. Das Ergebnis der Subtraktion steht immer in dem Register, das hinter dem Komma angegeben ist. Hierbei ist darauf zu achten, ob es sich bei der Subtraktion um Langworte, Worte oder Bytes handelt.

- **BNE:** Dieser Befehl überprüft, ob das Ergebnis einer Operation Null ergeben hat. So lange die Operation ungleich null ist, wird zu dem hinter dem Befehl angegeben Label gesprungen. Ist das Ergebnis gleich Null, wird der nächste Befehl abgearbeitet.

- Des weiteren haben wir gelernt, daß es sinnvoll sein kann, Variablen einen Namen zu geben. Dadurch werden Programme verständlicher, und zum anderen ist es dadurch einfacher, die Werte der Variablen zu ändern.

- Weiterhin haben wir gelernt, Programme vernünftig zu strukturieren, also in Haupt- und Unterprogramm aufzuteilen, um eine bessere Übersicht zu haben. Unterprogramme werden mit dem Befehl JSR und dem Namen des Unterprogramms aufgerufen. Einem Unterprogramm muß der Name, das Label, vorangestellt werden, dieser muß mit einem Doppelpunkt abgeschlossen sein. Um ein Unterprogramm wieder zu verlassen, benötigen wir den Befehl RTS.

- Anhand des Programms haben wir auch gelernt, was Schleifen sind und wie sie auszusehen haben. Um Schleifen zu konstruieren, muß ein Anfangswert vorgegeben werden, dieser muß durch Addition oder Subtraktion verändert werden. Des weiteren muß ein Endwert festgelegt werden, mit dem der Zähler verglichen werden kann.

4. Weitere Anwendungen - die Maus

Nachdem wir nun schon einige Befehle kennengelernt haben und auch schon ein recht umfangreiches Programm geschrieben haben, wollen wir in diesem Kapitel weitere Befehle vorstellen und ein wenig mit unserer Maus herumspielen. Des weiteren wollen wir Ihnen eine Möglichkeit zeigen, wie Sie aus fehlerhaften Programmen, z.B. aus Endlosprogrammen ohne große Schwierigkeit wieder herauskommen.

4.1 Abbruch auf Tastendruck

Wir wollen jetzt ein Programm schreiben, aus dem Sie nur wieder herauskommen, in dem Sie die linke Maustaste drücken.

Wie erfährt man eine Betätigung der Maustaste?

Um so ein Programm zu schreiben, müssen wir natürlich wissen, an welcher Stelle im Amiga die Information für die linke Maustaste abgelegt ist. Diese Information liegt in derselben Speicherstelle wie die Information über die LED, nur ist ein anderes Bit für diese Information zuständig, nämlich das siebte Bit, also das mit der Nummer 6. Dieses Bit ist auf 1 gesetzt, so lange die linke Maustaste nicht gedrückt ist, und auf 0, sobald sie gedrückt ist.

Damit wissen wir also, welches Bit welcher Speicherstelle wir überprüfen müssen, um einen Abbruch des Programms mit Hilfe der linken Maustaste zu erreichen, das siebte Bit der Speicherstelle $BFE001.

Wir können unser Programm jetzt fast schon schreiben, wir müssen nur noch einen neuen Befehl dazu kennenlernen und zwar den Befehl:

```
BTST
```

Der Befehl BTST

Mit diesem Befehl kann man jedes einzelne Bit einer Speicherstelle prüfen, also feststellen, ob es gleich eins oder gleich null ist. Dazu muß hinter dem Befehl als erstes die Nummer des Bits, das geprüft werden soll stehen und hinter dem Komma die Speicherstelle, in der dieses Bit geprüft werden soll. Wollen wir nun prüfen, ob die Maustaste gedrückt ist oder nicht, so müßte eine Zeile in unserem Programm lauten:

```
BTST #06,$BFE001
```

Bei der Überprüfung wird das Zeroflag gesetzt, also ein spezielles Bit im Statusregister des Prozessors. Ist das zu überprüfende Bit gleich eins, so wird das Zeroflag auf null gesetzt, ist es gleich null, so wird das Zeroflag auf eins gesetzt. Somit kann man nach einer solchen Überprüfung wieder den Befehl BNE benutzen, um zu sehen, ob das Zeroflag gesetzt ist oder nicht.

Kommen wir zu dem Programm, in dem die Dinge, die wir in dem vorherigen Kapitel gelernt haben, auch wieder anwenden wollen. Zum einen wollen wir den Variablen in diesem Programm einen Namen geben, und zum anderen wollen wir wieder eine Schleife konstruieren. Auch wollen wir das Programm zur besseren Übersicht wieder mit Kommentaren versehen.

Ein Testprogramm für die linke Maustaste

Unser Programm sieht folgendermaßen aus:

```
START:

; ***** Variablen setzen *****
Basis = $BFE001
Maus  = 06

; ***** Hauptprogramm *****
SCHLEIFE:
   BTST #Maus,Basis
   BNE SCHLEIFE
ILLEGAL
```

Aktivieren Sie den Editor mit <Esc>, und geben Sie das Programm ein. Schalten Sie dann wieder in den Befehls-Modus, assemblieren Sie das Programm, und starten Sie es. Wenn Sie die Frage nach den Breakpoints ignoriert haben und die Eingabe-Taste gedrückt haben, sehen Sie auf Ihrem Bildschirm, daß im Befehls-Modus keine neue Meldung erscheint, der SEKA sich nicht wieder betriebsbereit meldet. Das liegt daran, daß unser Programm immer noch abgearbeitet wird und der SEKA die Kontrolle erst wieder erhält, wenn Sie die linke Maustaste gedrückt haben. Machen Sie dies also, und Sie sehen, es erfolgt die Registerausgabe, und die Kontrolle ist wieder beim SEKA.

Wir wollen Ihnen an dieser Stelle noch einmal erklären, wie dieses Programm funktioniert, wie es also im einzelnen abgearbeitet wird.

- Als erstes wird, wie in jedem Programm, ein Anfangslabel gesetzt, um die Startadresse des Programms leicht herausfinden zu können. Wir haben dieses Label wieder Start genannt.

- Als nächstes haben wir die Variablen gesetzt, die wir für dieses Programm benötigen. Zum einen ist dies die Speicherstelle, in der sich die Information über die linke Maustaste befindet. Diese Variable haben wir BASIS genannt. Dann haben wir noch eine Variable definiert, und zwar die Nummer des Bits, das sich beim Betätigen der linken Maustaste ändert. Diese Variable haben wir Maus genannt.

- Danach geht es direkt ins Hauptprogramm, das nur aus einer Schleife besteht, in der abgefragt wird, ob die linke Maustaste gedrückt ist oder nicht. Wie funktioniert nun diese Schleife?

- Mit dem Befehl BTST #6,$BFE001 wird überprüft, ob das siebte Bit der Speicherstelle (also das mit der Nummer 6) gleich eins oder null ist. Je nachdem wird dann das Zeroflag gesetzt.

- Mit dem Befehl BNE wird anschließend noch überprüft, ob das Zeroflag gesetzt ist oder nicht. Danach wird entschieden, ob die Schleife erneut durchlaufen oder ob der nächste Befehl abgearbeitet wird.

- Wenn der Prozessor die Schleife verläßt, trifft er auf ein ILLEGAL, wodurch die Kontrolle wieder dem SEKA übergeben wird.

Nun werden Sie sich sicher fragen, wofür so ein Programm überhaupt notwendig ist. Nun, wir haben Ihnen hier einen Vorschlag anzubieten.

- Bauen Sie dieses Programm doch als Unterroutine in Ihre Programme ein, so haben Sie immer die Möglichkeit, aus Ihrem Programm herauszukommen, selbst wenn Sie mal Fehler in Ihrem Programm haben sollten, die ein Verlassen des Programms verhindern (eine sogenannte Endlosschleife).

4.2 So stellt man Mausbewegungen fest

Wir wollen nun ein weiteres Programm schreiben. In diesem Programm wollen wir die Sachen, die wir bisher gelernt haben, kombinieren und noch ein paar zusätzliche Dinge kennenlernen.

Das Programm, was wir schreiben wollen, soll auf die Bewegungen der Maus reagieren, und zwar insofern, daß bei jeder Mausbewegung die LED an- bzw. ausgehen soll.

Wir werden in diesem Programm ein paar neue Befehle benötigen, die wir schon vorab ein wenig erläutern möchten.

Der Befehl CMP

Mit diesem Befehl können Sie die Größe zweier Operanden vergleichen. Dies funktioniert folgendermaßen: Zunächst wird der Quelloperand vom Zieloperanden subtrahiert. Je nach Ergebnis

des Vergleichs wird wieder das Zeroflag gesetzt. Ergibt die Subtraktion Null, wird das Zeroflag auf eins gesetzt, ansonsten wird es auf null gesetzt.

Hinweis: Zwar wird die Quelle vom Ziel subtrahiert, doch werden dabei weder Quelle noch Ziel geändert, da diese Subtraktion intern im Prozessor abläuft.

Der Befehl BEQ

Dieser Befehl ist dem Befehl BNE sehr ähnlich. Er gehört auch zur gleichen Kategorie von Befehlen und zwar zur Kategorie der Befehle, die bedingte Verzweigungen auslösen. Mit dem Befehl BEQ erreichen Sie nun das Gegenteil wie mit dem Befehl BNE. Dieses erkennt man schon, wenn man sich anschaut, wofür diese Abkürzung steht. BEQ ist die Abkürzung für BRANCH EQUAL, was übersetzt soviel bedeutet, wie: Verzweige, wenn gleich.

Dieser Befehl überprüft wieder den Zustand des Zeroflags. Ist das Zeroflag gesetzt, also gleich eins, so verzweigt der Befehl zum angegebenen Label, ansonsten wird mit dem nächsten Befehl fortgefahren.

Der Befehl JMP

Mit diesem Befehl können Sie Label anspringen, wie die Übersetzung von JUMP: "Springe" schon andeutet. Im Gegensatz zum JSR merkt sich der Prozessor jedoch nicht, von welcher Stelle aus der Sprung erfolgte. Es kann also nicht wieder zurückgesprungen werden. Der Befehl entspricht dem GOTO von BASIC.

Der Befehl BTST

Mit diesem Befehl können Sie den Zustand eines einzelnen Bits abfragen. Je nach Ergebnis wird wieder das Zeroflag gesetzt. Und zwar wieder entgegengesetzt zum Zustand des Bits. Ist das Bit also auf null gesetzt, wird das Zeroflag auf eins gesetzt und umgekehrt. Die Abfrage funktioniert folgendermaßen:

Hinter dem Befehl muß zunächst die Bitnummer des Bits angegeben werden und dann die Speicherstelle, in der das Bit überprüft werden soll. Der Zustand des Bits läßt sich wieder mit einem BRANCH-Befehl überprüfen.

Der Befehl BCHG

Mit diesem Befehl können Sie den Zustand eines Bits erfahren und ihn ändern. Der ehemalige Zustand des Bits wird im Zeroflag abgelegt, und zwar wieder umgekehrt zum Zustand des Bits. Dann wird der Zustand des Bits invertiert, also genau umgedreht: War das Bit eins, wird es null, und war es null, wird es nun eins. Die Änderung erfolgt folgendermaßen:

Die Nummer des Bits, das geändert werden soll, muß direkt hinter dem Befehl stehen. Danach muß die Speicherstelle folgen, deren Bit geändert werden soll.

Nachdem wir nun die benötigten neuen Befehle kennengelernt haben, wollen wir Ihnen das Programm zeigen, das die Mausbewegung auf die LED überträgt und erst abgebrochen wird, wenn die linke Maustaste gedrückt wird.

```
START:

; **** Variablen setzen ****
Basis   = $BFE001            ; Speicherstelle für LED
Mausbew = $DFF00A            ; Speicherstelle für Mausbewegung
Mausta  = 06                 ; linke Maustaste
Warte   = 100000             ; Zähler für Warteschleife
Falsch  = 0                  ; Rückgabewert von Unterprogrammen
Wahr    = 1                  ; Rückgabewert von Unterprogrammen

; **** Initialisierung ****
MOVE Mausbew,D5              ; Mausposition merken

; **** Hauptprogramm ****
HAUPTSCHLEIFE:
  JSR ABBRUCH_TEST           ; Abbruch prüfen
    CMP.L #Wahr,D0           ; erwünscht ?
    BEQ ENDE                 ; Ja, Ende
  JSR MAUS_TEST              ; Mausbewegung prüfen
    CMP.L #Wahr,D0           ; bewegt?
    BNE HAUPTSCHLEIFE_ENDE; nein, weiter
  JSR LED                    ; ja, LED blinken lassen
```

```
HAUPTSCHLEIFE_ENDE:
  JMP HAUPTSCHLEIFE        ; zurück zum Anfang
ENDE:                      ; Programm beenden
  MOVE.B #00,Basis         ; dabei LED einschalten
  ILLEGAL                  ; Abbrechen

; **** Abbruchprogramm ****
ABBRUCH_TEST:
  MOVE.L #Falsch,D0        ; Rückgabewert vorgeben
  BTST #Mausta,Basis       ; Maustaste gedrückt ?
  BNE ABBRUCH_TEST_ENDE    ; nein, Testende, sonst:
  MOVE.L #Wahr,D0          ; Rückgabewert ändern
ABBRUCH_TEST_ENDE:
  RTS                      ; zurück zum Hauptprogramm

; **** Überprüfung der Mausbewegung ****
MAUS_TEST:
  MOVE.L #Falsch,D0        ; Rückgabewert vorgeben
  MOVE Mausbew,D6          ; Neue Mausposition holen
  CMP D6,D5                ; Vergleich der Mauspositionen
  BEQ MAUS_TEST_ENDE       ; gleich, Test Ende, sonst:
  MOVE D6,D5               ; Neue Mausposition merken
  MOVE.L #Wahr,D0          ; Rückgabewert ändern
MAUS_TEST_ENDE:
  RTS                      ; Rückkehr zum Hauptprogramm

; **** Ein- bzw. Ausschalten der LED ****
LED:
  BCHG #01,Basis           ; LED invertieren
  MOVE.l #Warte,D0         ; Zähler für Warteschleife vorgeben
WARTESCHLEIFE:
  SUB.L #01,D0             ; Zähler erniedrigen
  BNE WARTESCHLEIFE        ; Nicht Null, dann Warten, sonst:
  RTS                      ; Rückkehr zum Hauptprogramm
```

Nachdem Sie dieses etwas längere Programm nun eingegeben haben, aktivieren Sie den Befehls-Modus. Nach dem Assemblieren, wobei bei Ihnen hoffentlich keine Fehlermeldung auftritt, können Sie dieses Programm nun starten.

Hinweis: Bevor Sie ein eingegebenes und fehlerfrei assembliertes Programm starten, sollten Sie es IMMER speichern. Nichts ist ärgerlicher als ein Absturz auf Grund eines Tippfehlers, wenn das Programm noch nicht gespeichert ist.

Wenn Sie nun Ihre Maus bewegen, sehen Sie, wie die LED darauf reagiert. Sie geht aus bzw. an. Bewegen Sie die Maus nicht,

so passiert auch nichts. Sie befinden sich nun in einer Endlosschleife, die Sie so lange nicht verlassen können, bis Sie die linke Maustaste gedrückt haben. Drücken Sie die linke Maustaste, wird die Kontrolle wieder dem SEKA übergeben. Sie erkennen dies an der bekannten Registeranzeige, die sofort erscheint.

Wie funktioniert dieses Programm also? Schauen wir es uns doch noch einmal an:

Das erste, was wir in diesem Programm getan haben, ist die Definition der Variablen. Zu ihnen gehören:

Basis Dies ist die Speicherstelle, in der die Informationen zur LED gespeichert sind, und zwar ist das Bit mit der Nummer eins für diese Information zuständig. Des weiteren ist in dieser Speicherstelle die Information abgelegt, ob die linke Maustaste gedrückt ist oder nicht. Für diese Information ist das Bit mit der Nummer sechs zuständig.

Mausta Mit dieser Variablen definieren wir die Nummer des Bits, in welchem die Information abgelegt ist, ob die Maustaste gedrückt ist oder nicht.

Mausbew Dies ist die Speicherstelle, in der die Information abgelegt wird, ob die Maus bewegt wurde oder nicht.

Falsch Dies wird als Rückgabewert für Unterroutinen benutzt. Hat etwas im Unterprogramm nicht richtig funktioniert, darf dieses ja nicht einfach mit einem ILLEGAL enden. Statt dessen liefert es dann an das Hauptprogramm den Wert "Falsch" zurück, so daß das Hauptprogramm darauf reagieren kann.

Wahr Genauso wie Falsch, Rückgabewert einer Unterroutine. Falsch und Wahr sind viel aussagekräftigere Bezeichnungen im Programm als 0 oder 1. Nutzen

Sie auch in eigenen Programmen möglichst viele und aussagekräftige Variablen, damit keine Mißverständnisse passieren und Sie Ihr Programm auch noch nach vier Wochen verstehen können.

Damit ist die Definition der Variablen abgeschlossen.

Wir kommen nun zur Initialisierung, was etwas Neues für uns ist. Bei einer Initialisierung werden Registern oder Speicherstellen zu Beginn des Programms bestimmte Werte zugewiesen. Man bestimmt damit also einen Anfangswert. Dies ist beispielsweise wichtig, wenn man einen neuen Wert mit einem älteren vergleichen will. Beim ersten Starten hat man ja noch keinen "älteren Wert".

Nach der Initialisierung kommt das Hauptprogramm. Dies ist wieder so aufgebaut, daß es Unterroutinen aufruft. Diesmal allerdings nicht in jedem Fall, sondern in diesem Programm kommt es darauf an, welche Rückgabewerte von den jeweiligen Unterprogrammen zurückgeliefert werden. Die Verzweigung in die Unterprogramme ist somit eine bedingte Verzweigung. Die Unterprogramme, in die hineingesprungen wird, kennen wir fast alle. Da ist einmal das Unterprogramm ABBRUCH_TEST, das überprüft, ob die linke Maustaste gedrückt wurde.

Als erstes geben wir eine Vorgabe für den Rückgabewert aus diesem Unterprogramm. Nun wird mit dem Befehl BTST überprüft, ob die Maustaste gedrückt wurde oder nicht. Wurde die Maustaste gedrückt, so wird direkt zum Ende dieses Unterprogramms gesprungen, ansonsten wird der Rückgabewert dieses Unterprogramms geändert und dann erst ans Ende des Unterprogramms gesprungen. Das Ende dieses Unterprogramms ist der Befehl RTS, mit dem wieder ins Hauptprogramm zurückgekehrt wird.

Zwei Besonderheiten dieses Unterprogramms wollen wir noch einmal kurz festhalten, weil wir Sie immer wieder verwenden werden:

1. Wenn Unterprogramme am Ende ein Label erhalten, vergeben wir den Namen des Unterprogramms mit dem Zusatz "ENDE". Dadurch wissen wir immer, zu welchem Unterprogramm unser Label gehört.

2. Wenn ein Unterprogramm eine Situation überprüft und dementsprechend einen Wert zurückliefert, gibt das Programm den wahrscheinlichen Wert vor, so daß schon mit dem richtigen Wert zum Endlabel gesprungen werden kann. In unserem Fall nimmt das Unterprogramm die sehr häufig eintretende Situation an, daß die Maustaste nicht gedrückt wurde, um in diesem Fall sofort enden zu können. Ist die Taste gedrückt, muß dann natürlich der Rückgabewert geändert werden.

Im Hauptprogramm wird nun zunächst einmal der Rückgabewert der Unterroutine überprüft, und je nach Rückgabewert wird verzweigt. Entweder zum Ende des gesamten Programms oder in die nächste Unterroutine MAUS_TEST.

Die nächste Unterroutine überprüft, ob die Maus bewegt wurde oder nicht. Hier wird auch zunächst wieder eine Vorgabe für den Rückgabewert gemacht. Danach wird die neue Position der Maus in ein Datenregister übertragen und dann mit Hilfe des Befehls CMP mit der vorherigen Position der Maus verglichen. Wird dieses Unterprogramm zum erstenmal aufgerufen, wird mit dem Initialisierungswert verglichen.

Sind die Mauspositionen gleich, wird direkt ans Ende dieses Unterprogramms gesprungen, ansonsten wird die neue Mausposition festgehalten und der Rückgabewert geändert, und erst danach wieder mit RTS ins Hauptprogramm zurückgesprungen.

Im Hauptprogramm wird nun erst einmal wieder der Rückgabewert geprüft, und je nach Rückgabewert verzweigt. Entweder, wenn sich der Rückgabewert nicht geändert hat, an den Anfang des Hauptprogramms, oder in das nächste Unterprogramm LED.

Das nächste Unterprogramm ist uns bekannt. Mit Hilfe dieses Programms wird die LED aus- bzw. eingeschaltet. Die einzige Änderung, die wir hier vorgenommen haben, ist, daß wir uns einen neuen Befehl zur Hilfe genommen haben und zwar den Befehl BCHG, mit dem wir direkt den Zustand eines Bits ändern können. Wir brauchen daher erst gar nicht den Zustand des Bits zu überprüfen, sondern können das Bit direkt ändern. Nach dem Abarbeiten des Unterprogramms gehen wir mit RTS wieder ins Hauptprogramm zurück.

Im Hauptprogramm sind wir nun am Ende angelangt und springen mit JMP wieder an den Anfang des Hauptprogramms zurück. Dieses wird erneut abgearbeitet und zwar so lange, bis bei einer Überprüfung der linken Maustaste festgestellt wird, daß diese gedrückt wurde.

Zusammenfassung

Wir wollen noch einmal zusammenfassen, was wir in diesem Kapitel alles gelernt haben.

- Da sind zunächst einmal ein paar neue Befehle:

- Mit dem Befehl BTST können wir gezielt Informationen über den Zustand eines bestimmten Bits erhalten. Dazu müssen wir als erstes die Nummer des Bits wissen und dann hinter dem Komma die Speicherstelle des Bits angeben, welches zu überprüfen ist. Die Information über den Zustand wird im Zeroflag abgelegt.

- Mit dem Befehl CMP ist es uns möglich, den Inhalt von Registern zu vergleichen. Dabei wird der Inhalt des Zieloperanden vom Quelloperanden abgezogen. Je nachdem, ob das Ergebnis gleich null ist oder nicht, wird das Zeroflag gesetzt und zwar umgekehrt wie das Ergebnis.

- Der Befehl BEQ ist fast analog zu dem schon vorher kennengelernten Befehl BNE. Bei ihm wird der Zustand des Zero-

flag überprüft. Ist das Zeroflag gleich null, so wird verzweigt, ansonsten wird mit dem nächsten Befehl fortgefahren.

- Der Befehl JMP ist ein Sprungbefehl, der zu dem angegebenen Ziel springt. Es kann hier nicht wie bei JSR ein Rücksprung erfolgen.

- Der Befehl BCHG ändert den Zustand eines bestimmten Bits einer Speicherstelle. Dazu muß zunächst die Nummer des Bits, das geändert werden soll, angegeben werden und danach, hinter dem Komma die Speicherstelle, zu der dieses Bit gehört.

- Des weiteren haben wir etwas über Initialisierungen in Programmen gelernt. Wir wollten ja immer den aktuellen Wert der Mausposition mit dem vorherigen vergleichen. Dazu mußte vor dem eigentlichen Hauptprogramm ein Startwert bestimmt werden. Eine solche Initialisierung ist also günstig, um Startwerte für ein Programm festzulegen.

- Wir haben auch gelernt, daß es günstig ist, in Unterprogrammen mit Rückgabewerten zu arbeiten. So kann dann im Hauptprogramm je nach Rückgabewert verzweigt werden.

- Des weiteren haben wir die sinnvolle Anwendung unseres Abbruchprogramms kennengelernt. Mit ihm ist es uns möglich, aus Endlosprogrammen herauszukommen und zwar zu dem Zeitpunkt, an dem wir es wünschen.

5. Das Betriebssystem hilft weiter

In diesem Kapitel wollen wir nun die Voraussetzungen für etwas umfangreichere Projekte schaffen. Damit unsere Programme dadurch aber nicht zwanzig Seiten lang werden, wollen wir uns kostenlos und ohne große Anstrengungen eine riesige Bibliothek nützlicher Hilfsprogramme verschaffen.

5.1 Wozu braucht man ein Betriebssystem

In den letzten Kapiteln haben wir schon einiges über Unterroutinen gelernt. Mit Unterroutinen gestalten wir ein Programm übersichtlicher, und es erleichtert uns weiterhin die Arbeit bei neuen Programmen, weil wir Unterroutinen gesondert abspeichern und danach beliebig in andere Programme einbauen können. Wir konnten jeweils mit einem JSR in die Unterprogramme hineinspringen und diese mit RTS wieder verlassen. Unterprogramme haben uns somit die Arbeit beim Programmieren sehr erleichtert.

Nun zu einer großen Überraschung. Im Betriebssystem des Amigas sind schon sehr viele nützliche Unterprogramme enthalten. Es gibt somit viele Unterroutinen, die wir nutzen können, ohne sie vorher selbst zu programmieren. Dies ist in vielen Punkten eine große Erleichterung für uns. Sie werden nun sicherlich neugierig sein und sich fragen, wie man denn an diese Unterroutinen gelangt. Dies ist etwas schwieriger, als an unsere eigenen Unterroutinen heranzukommen. Um zu lernen, wie man mit betriebsinternen Unterroutinen umgeht, müssen wir Ihnen erst noch ein paar Erklärungen liefern.

5.2 Offsets und Bibliotheken - was ist das

Hier wollen wir Sie mit ein paar neuen Begriffen vertraut machen, die Sie bestimmt schon gehört haben, wenn Sie sich schon

etwas länger mit dem Amiga beschäftigen. Wir wollen nun einmal etwas näher erklären, was es mit diesen Begriffen auf sich hat.

Bibliothek

Eine Bibliothek, auch Library genannt, ist eine Sammlung von Unterroutinen. In so einer Bibliothek werden Unterroutinen archiviert, die zu ein und demselben Gebiet gehören. So haben diese Bibliotheken denn auch Namen, an denen man den Zweck der jeweiligen Bibliothek erkennen kann. Es gibt also beispielsweise ein DOS-Bibliothek, eine Grafik-Bibliothek usw.

Offset

Um Ihnen zu erklären, was ein Offset ist, müssen wir ein wenig ausholen. Sämtliche Amigas sind so konstruiert, daß sie nur eine "feste" Speicherstelle haben. Dies ist die Speicherstelle, in der die Adresse der Exec.library liegt. Diese Speicherstelle für die Exec.library ist nun wirklich für alle Amigas gleich. Sie liegt in der Speicherstelle 4.

Alle anderen Adressen können nun bei verschiedenen Amigas mit verschiedenen Betriebssystemversionen an unterschiedlichen Stellen liegen. Sie können für Ihren Amiga zwar immer feststellen, an welcher Stelle welche Adresse abgelegt ist, aber auf einem anderen Amiga kann die Adresse an einer ganz anderen Stelle liegen.

Damit Ihre Programme sicher auf allen Amigas funktionieren, müssen Sie sich an bestimmte Spielregeln halten. Wenn Sie dies tun, ist sichergestellt, daß Ihre Programme auf allen Amigas funktionieren. Diese Vorgehensweise werden wir in diesem Kapitel kennenlernen.

Hinweis: Zugegebenermaßen haben wir uns bei unseren bisherigen Beispielprogrammen nicht immer an diese Regeln gehalten. Wir bitten hiermit um Entschuldigung. Allerdings wären unsere Programme sonst erheblich länger und schwerer zu verstehen.

Basisadresse einer Bibliothek

Wenn Sie nun eine Bibliothek öffnen wollen, müssen Sie sich zunächst über den Namen der Bibliothek die Basisadresse holen. Wenn Sie diese Basisadresse haben, können Sie mit Hilfe dieser Basisadresse sämtliche Unterprogramme der Bibliothek aufrufen. Sämtliche Unterroutinen liegen nämlich immer relativ zu der Basisadresse.

Im Amiga ist dies so geregelt, daß die Unterroutinen im Speicher immer vor der Basisadresse liegen. Da Sie relativ zur Basisadresse liegen, muß man um in diese Unterroutinen einzuspringen, also nur noch die Differenz zwischen Basisadresse und Adresse der Unterroutine angeben. Die Abstände zwischen Basisadresse und Unterroutinen sind immer gleich.

Diese Abstände nennt man die Offsets der Unterroutinen. Da die Adressen der Unterroutinen vor der Basisadresse liegen, ist klar, daß die Offsets immer einen negativen Wert haben.

5.3 Eine Bibliothek ist immer da - Exec.library

Wie schon oben erwähnt, ist die Adresse der Exec.library die einzige Adresse, die in jedem Amiga an der gleichen Stelle abgelegt ist. Sie ist auch die einzige Library, die Sie nicht öffnen müssen, ja auch eigentlich gar nicht öffnen könnten, da die Unterroutine zum Öffnen von Librarys in der Exec.library selbst vorhanden ist. Die Exec.library ist somit eine Bibliothek, die Ihnen immer und jederzeit zur Verfügung steht.

Das Öffnen einer Bibliothek läuft also immer nach folgendem Schema ab:

1. Über die spezielle Speicherstelle 4 ermittelt man die Basisadresse der Exec.library.

2. Über diese Basisadresse kann man die Funktion zum Öffnen einer neuen Library verwenden. Dieser Funktion übergibt man den Namen der Library, die geöffnet werden soll.

3. Die Funktion liefert als Rückgabewert die Basisadresse der neuen Library. Über diese Basisadresse kann man die gewünschten Unterprogramme aufrufen.

5.4 Bibliothek öffnen und eine Funktion aufrufen

Doch kommen wir nun zu unserem nächsten Programm. Mit diesem Programm wollen wir Ihnen zeigen, wie man eine Bibliothek öffnet.

Wir wollen hier die Dos.library öffnen, weil wir später auch mit ihr arbeiten wollen. Um die Dos.library zu öffnen, benötigen wir zunächst einmal das Offset der Routine OpenLibrary der Exec.library zum Öffnen von Bibliotheken. Der Wert dieses Offsets ist:

```
-$0228 (Hex-Schreibweise)
```

Des weiteren benötigen wir den Namen der Dos.library, um diesen Namen an das Unterprogramm zu Öffnen der neuen Library zu übergeben.

Dies machen wir mit Hilfe einer sogenannte Assembler-Directive, das ist eine Anweisung an den Assembler und kein Befehl für den Prozessor. Die Assembler-Directive mit dem Namen der Bibliothek muß im Programm folgendermaßen aussehen:

```
Dosname:
DC.B `dos.library`,0
```

Die beiden Buchstaben "DC" stehen für Define Constant (definiere eine Konstante), das ".B" steht wie gewohnt für Byte. Das Hochkomma "`" erhalten Sie mit der Taste über der Tabulator-Taste.

Hinweis: Sie können statt des Hochkommas beim SEKA auch die Anführungszeichen (") verwenden.

Wichtig ist weiterhin, daß Sie den Namen der zu öffnenden Bibliothek in kleinen Buchstaben eingeben und wie im Beispiel mit ",0" beenden. Wählen Sie eine andere Schreibweise, wird die Bibliothek nicht geöffnet.

Die Unterroutine OpenLibrary der Exec.library erwartet nun noch bestimmte Optionen in bestimmten Registern. So wird der Name der Library im Adreßregister A1 und die Version der Library im Datenregister D1 erwartet. Zur "Version" nur soviel: So, wie das Betriebssystem immer weiterentwickelt und verbessert wird und dann eine neue, höhere Versionsnummer erhält, so gilt dies auch für die Bibliotheken. Es kann beispielsweise sein, daß eine bestimmte Funktion erst ab einer bestimmten Versionsnummer vorhanden ist.

Gibt man nun eine Versionsnummer in D1 an, wird die Library nur dann geöffnet, wenn die Versionsnummer der vorhandenen Bibliothek gleich der angegebenen oder sogar höher ist. Gibt man als Versionsnummer eine 0 an, ist jede mögliche Library-Version größer, und die Bibliothek wird unabhängig von der Version geöffnet.

Nachdem Sie diese Optionen übergeben haben, können Sie die Unterroutine zum Öffnen einer Bibliothek aufrufen. Hat das Öffnen der Bibliothek funktioniert, so wird Ihnen die Basisadresse im Register D0 zurückgeliefert. Ansonsten erhalten Sie in D0 eine Null. Es ist nun wichtig, das Register D0 zu überprüfen, bevor der Rest des Programms abgearbeitet wird. Hat das Öffnen der Bibliothek nämlich nicht funktioniert, sollte direkt zum Ende des Programms gesprungen werden.

Auf keinen Fall darf mit der Basisadresse 0 irgendeine Funktion der Library aufgerufen werden. Das führt sicher zur Guru Meditation.

Weiterhin ist es noch wichtig, eine geöffnete Bibliothek wieder zu schließen, wenn sie nicht mehr benötigt wird. Das gilt übrigens für alles, was Sie sich vom Betriebssystem geben lassen: Geben Sie es zum Ende des Programms wieder zurück.

In dem nun folgenden Programm zeigen wir Ihnen, wie die Dos.library geöffnet wird.

Das Programm zum Öffnen der Dos.library

```
Start:

; **** Variablen definieren ****
Execbase = 4                 ; Adresse der Exec.library
OpenLibrary = -$0228         ; Offset
CloseLibrary = -$019E        ; Offset
Delay = -$00C6               ; Offset zum Warten

; **** Hauptprogramm ****
JSR OpenDos
CMP.L #0,D0                  ; Hat Öffnen geklappt?
BEQ Ende                     ; nein, Programmende!
JSR Warten
JSR CloseDos
JMP Ende

; **** Dos.library öffnen ****
OpenDos:
MOVE.L Execbase,A6           ; Basisadresse nach A6
MOVE.L #Dosname,A1           ; Libraryname nach A1
MOVE.L #0,D0                 ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)          ; Interne Unterroutine
MOVE.L D0,Dosbase            ; sonst: Adresse merken
RTS

; **** Warteschleife ****
Warten:
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
MOVE.L #100,D1               ; Wartezeit nach D1
JSR Delay(A6)                ; Warteroutine 1/50 Sek.
RTS

; **** Dos.library schließen ****
CloseDos:
MOVE.L Dosbase,A1            ; Dos-Basisadresse nach A1
MOVE.L Execbase,A6           ; Basisadresse nach A6
JSR CloseLibrary(A6)         ; Interne Unterroutine
RTS
; **** Programm beenden ****
Ende:
```

```
ILLEGAL

; **** Speicherreservierung ****

Dosname:
DC.B `dos.library`,0      ; Name der Dos.library

EVEN
Dosbase:
DC.L 0                    ; Platz für Dos-Basisadresse
```

Erklärung des Programms

Wir wollen Ihnen an dieser Stelle erklären, wie dieses Programm funktioniert, und welche Dinge wichtig sind damit es funktioniert.

Was haben wir in diesem Programm gemacht:

Der erste Block, den unser Programm enthält, ist wie immer der Block, in dem die Definition der Variablen stattfindet. Die hier definierten Variablen sind:

1. Die Basisadresse der Exec.library

2. Die verschieden Offsets der benötigten Unterroutinen
   - Offset zum Öffnen einer Bibliothek
   - Offset zum Schließen einer Bibliothek
   - Offset für eine Verzögerungsroutine

Der nächste Block ist das Hauptprogramm, von dem aus die verschiedenen Unterroutinen aufgerufen werden.

Nun folgen die einzelnen Unterroutinen, diese wollen wir Ihnen etwas näher erklären.

Die Unterroutine OpenDos

Mit dieser Unterroutine wird, wie der Name schon sagt, die Dos.library geöffnet. Um die Dos.library zu öffnen, sind einige Vorarbeiten nötig. Der Prozessor benötigt zunächst einmal die Basisadresse der Library, von der aus eine betriebsinterne Un-

terroutine aufgerufen werden soll, im Register A6. Da wir eine Unterroutine der Exec.library aufrufen wollen, nämlich die Unterroutine OpenLibrary, müssen wir die Basisadresse der Exec. library nach A6 übertragen. Dies geschieht mit Hilfe der Programmzeile:

```
MOVE.L Execbase,A6
```

Des weiteren müssen wir noch einige Angaben in gewisse Register übertragen. Zum einen den Namen der Bibliothek, die wir öffnen wollen, zum anderen die Versionsnummer der zu öffnenden Bibliothek. Den Namen der zu öffnenden Bibliothek müssen wir zunächst einmal am Ende des Programms ablegen. Wie dies funktioniert, haben wir Ihnen schon oben gezeigt. Man benötigt dafür im Programm die Zeilen:

```
Dosname:
DC.B `dos.library`,0
```

Wie schon gesagt, ist es wichtig, daß der Name der zu öffnenden Bibliothek in kleinen Buchstaben eingegeben wird. Des weiteren ist es wichtig, den Namen mit einer Null abzuschließen, damit die Unterroutine der Exec.library weiß, wo der Name der zu öffnenden Bibliothek endet. Die Adresse, in der der Name abgelegt wurde, müssen Sie dem Register A1 übergeben. Die Versionsnummer wird in Register D0 erwartet. Geben Sie als Versionsnummer eine Null ein, so bedeutet dies, daß es sich um eine beliebige Version handeln kann. Nachdem wir die nötigen Vorbereitungen getroffen haben, können wir die betriebsinterne Unterroutine aufrufen und zwar mit der Zeile:

```
JSR OpenLibrary(A6)
```

Mit diesem Befehl geben Sie dem Amiga die Sprungadresse an, in der die Unterroutine OpenLibrary beginnt. Dies geschieht mit Hilfe einer besonderen Adressierungsart. Diese Adressierung wird "Adreßregister indirekt mit Adreßdistanz" genannt.

Was bedeutet dieser schöne Ausdruck? Es soll zu einer Adresse gesprungen werden, die eine bestimmte Distanz zum angegebe-

nen Adreßregister hat. In unserem Fall bedeutet das, daß zum Adreßregister, in dem die Basisadresse der Exec.library, mit dem Distanzwert, der genau unser Offset darstellt, gesprungen werden soll. Da unsere Offsets immer negativ sind wird also bei uns vom Adreßregister der Wert des Offset subtrahiert und dann an eben diese Stelle gesprungen.

Zum Schluß legen wir die Basisadresse der Dos.Library noch an einer bestimmten Speicherstelle (Dosbase) ab. Dazu wurde vorher mit der Assembler-Direktive DC.L 0 am Ende des Programms ein Langwort (vier Bytes) mit dem Wert 0 abgelegt. Dorthin kann nun die Basisadresse geschrieben werden.

Prüfen, ob Öffnen funktioniert hat

Danach müssen wir im Hauptprogramm prüfen, ob das Öffnen der Dos.library auch funktioniert hat. Denn nur wenn die Dos.library auch geöffnet wurde, wollen wir mit dem Programm fortfahren, ansonsten würden wir das Programm sofort beenden. Ob das Öffnen der Dos.library funktioniert hat oder nicht, kann anhand des Datenregisters D0 überprüft werden. Steht im Datenregister D0 eine Null, hat das Öffnen nicht funktioniert. Hat aber alles geklappt, wird im Datenregister D0 die Basisadresse der Dos.library zurückgeliefert. Wir vergleichen also den Wert in D0 mit der Zahl Null und verzweigen bei Übereinstimmung zum Programmende. Vielleicht ist Ihnen noch der Begriff "EVEN" davor aufgefallen. Direkt übersetzt bedeutet er "gerade" und stellt ebenfalls eine Assembler-Direktive dar. Vielleicht erinnern Sie sich noch daran, daß Langworte immer an einer geraden Adresse abgelegt werden müssen. Das stellen wir durch die Direktive EVEN sicher. Der Assembler füllt gegebenenfalls mit einem zusätzlich eingefügten Byte auf.

Die Verzögerungsroutine

Je nach Ergebnis der Überprüfung wird nun im Programm fortgefahren. Wir nehmen einmal an, daß das Öffnen der Dos.library funktioniert hat und also ins Hauptprogramm zurückgekehrt wird. Um nun schon nach außen hin sehen zu können, ob das Öffnen funktioniert hat oder nicht, wollen wir in

dem Fall, in dem es funktioniert hat eine Verzögerungsroutine durchlaufen lassen. Für diese Verzögerungsroutine nehmen wir uns ein betriebsinternes Unterprogramm der Dos.library zur Hilfe. Diese Unterprogramm hat den Namen Delay (Verzögern), und als Funktion wartet es sovielmal 1/50 Sekunden, wie vorher im Datenregister D1 festgelegt wurde.

Vom Hauptprogramm wird nun also in die Unterroutine "Warte" gesprungen. Da wir in dieser Unterroutine eine betriebssysteminterne Unterroutine der Dos.library aufrufen wollen, müssen wir zunächst einmal die Basisadresse ins Adreßregister A6 übertragen. Als weiteren Schritt müssen wir noch den Wartewert ins Datenregister D1 übertragen. Haben wir dies alles getan, können wir die Unterroutine Delay aufrufen. Dies geschieht wieder mit der oben erklärten Adressierungsart Adreßregister indirekt mit Distanzwert. Nach Aufruf dieser Unterroutine erfolgt wieder der Rücksprung ins Hauptprogramm.

Hinweis: Die betriebssysteminterte Unterroutine Delay ist in vielen Fällen einer selbstprogrammierten Warteschleife (Startwert in Register schreiben und so lange verringern, bis Null erreicht ist) vorzuziehen, weil während des Wartens keine Prozessorzeit verbraucht wird: Das mehr an Rechenzeit können also andere Tasks (Programme) verwenden und werden nicht unnötig gebremst.

Schließen der Dos.library

Als letzten Schritt des Hauptprogramms müssen wir die Dos.library noch schließen. Dies funktioniert ähnlich wie das Öffnen der Bibliothek. Zunächst müssen wir wieder die Basisadresse der Exec.library ins Adreßregister A6 übertragen. Die Unterroutine CloseLibrary erwartet nun noch die Basisadresse der zu schließenden Bibliothek in A1.

6. Wir öffnen ein eigenes Fenster

Nachdem wir nun erste Erfahrungen im Umgang mit dem Betriebssystem und den Unterroutinen der vorhandenen Libraries gemacht haben, wollen wir nun einige Routinen des Betriebssystems ausprobieren. Dazu werden wir ein einfaches Fenster öffnen und Texte in dieses Fenster ausgeben und Eingaben von der Tastatur verarbeiten.

6.1 So öffnet man ein DOS-Fenster

Bevor wir mit dem Öffnen eines Fensters beginnen, wollen wir noch auf zweierlei hinweisen:

1. Im Amiga ist nicht nur (fast) alles relativ, sondern beim Amiga kann man fast immer auf mehrere Weisen vorgehen: einfach und komplizierter. Meistens bietet dann zwar die kompliziertere Methode mehr Möglichkeiten, aber diese sind gar nicht immer nötig. Für das Öffnen eines Fensters bedeutet das: Wir können ein relativ einfaches DOS-Fenster öffnen, oder ein erheblich aufwendigeres Intuition-Fenster, das wir in einem späteren Kapitel kennenlernen werden.

   In diesem Kapitel werden wir das einfachere DOS-Fenster verwenden. Dies ist übrigens genauso ein Fenster, wie Sie es auch vom CLI her kennen - es enthält also beispielsweise kein Schließsymbol. Für unsere Zwecke reicht es aber völlig aus.

2. An dieser Stelle ist nun auch der Punkt gekommen, an dem Sie sich überflüssige Tipparbeit ersparen sollen. Die folgenden Programme werden nämlich immer wieder dieselben Unterroutinen enthalten, beispielsweise zum Öffnen und Schließen der Dos.library. Erstellen Sie am besten einmal einen Programmrumpf, der die entsprechenden Grundbe-

standteile eines Programms enthält, und laden Sie diesen, bevor Sie mit einem neuen Programm beginnen.

Speichern Sie außerdem Unterprogramme wie "Opendos", "Closedos" und "Warten" einzeln auf Diskette ab. Sie können diese "Bausteine" dann leicht in den Programmrumpf einfügen, in dem Sie den Cursor an die gewünschte Stelle bewegen und mit

```
R
```

und anschließender Angabe des Bausteinnamens das Unterprogramm an der aktuellen Cursor-Position einfügen.

Wie funktioniert ein DOS-Fenster?

Amiga-DOS ist nicht nur dazu da, Diskettenlaufwerke und deren Inhalt zu verwalten. Unter anderem kann DOS alle im Amiga vorhandenen Geräte verwalten, beispielsweise neben den Diskettenlaufwerken auch RAM:, PAR:, PRT: und SER. Ein etwas spezielles Gerät heißt CON:. CON fällt insofern etwas aus der Reihe, weil es eigentlich zwei Geräte darstellt: Bildschirm und Tastatur. Wird etwas zu dem Gerät CON: geschickt, geht diese Ausgabe auf den Bildschirm, werden Informationen vom Gerät geholt, wird dazu die Tastatur verwendet.

Da eine Ausgabe nicht einfach direkt auf den Bildschirm gehen kann, muß zusätzlich eine Fensterbeschreibung angegeben werden. Probieren Sie doch einmal die folgende Befehlszeile im CLI aus:

```
COPY S:Startup-Sequence CON:10/10/600/180/FENSTER
```

Anschließend öffnet sich auf dem Bildschirm ein neues Fenster, und in diesem wird die Startup-Sequence angezeigt. Sobald der Vorgang beendet ist, schließt sich das Fenster automatisch wieder. Die Zahlen hinter CON: geben übrigens die linke, obere Ecke des Fensters und die Breite und Höhe in Punkten an, der anschließende Text "FENSTER" wird von DOS als Titel für das Fenster verwendet.

Um Ihnen auch das Eingeben von Zeichen über CON: (also in diesem Fall über die Tastatur) zu demonstrieren, haben wir uns ein zweites Beispiel ausgedacht. Geben Sie doch einmal folgende Zeile vom CLI aus ein:

```
COPY CON:10/90/300/140/Eingabe CON:10/10/300/80/Ausgabe
```

Anschließend erscheinen zwei neue Fenster auf dem Bildschirm. Klicken Sie mit der Maus in das Fenster "Eingabe", um es zu aktivieren, und geben Sie einmal einige Zeichen von der Tastatur ein. Diese erscheinen sofort im Eingabe-Fenster, im Ausgabe-Fenster passiert nichts. Nun betätigen Sie zusätzlich die Taste Return. Sofort erscheinen die eingegebenen Zeichen im Ausgabe-Fenster.

Nun haben Sie sicher schon ein Gefühl dafür bekommen, wie man mit DOS und dem speziellen Gerät CON: ein Fenster für Ein- oder Ausgaben erstellen kann. Die beiden eben erstellten Fenster bekommt man übrigens vom Bildschirm, in dem man im Eingabe-Fenster CTRL + \ eingibt.

Das Programm: DOS-Fenster öffnen

Das folgende Programm öffnet ein DOS-Fenster, wartet zwei Sekunden und schließt es dann wieder.

```
Start:

; **** Variablen definieren ****
Execbase = 4                ; Adresse der Exec.library
Mode_Newfile = 1006         ; Dateimodus: Neue Datei erstellen

OpenLibrary = -$0228        ; Offset
CloseLibrary = -$019E       ; Offset
Open = -$001E               ; Offset
Close = -$0024              ; Offset
Delay = -$00C6              ; Offset zum Warten

; **** Hauptprogramm ****
JSR OpenDos                 ; Dos.library öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ Ende                    ; Nein, Ende
MOVE.L D0,Dosbase           ; sonst: Adresse merken
JSR OpenWin                 ; Fenster öffnen
CMP.L #0,D0                 ; Hat geklappt?
```

```
BEQ CloseDos              ; Nein, Ende
MOVE.L D0,Handle          ; sonst: Adresse merken
JSR Warten                ; 2 Sekunden warten
JSR CloseWin              ; Fenster wieder schließen
JSR CloseDos              ; Dos.library schließen, Ende
JMP Ende                  ; Programmende

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L #Dosname,A1        ; Libraryname nach A1
MOVE.L #0,D0              ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Dosbase,A6         ; Dos-Basisadresse nach A6
MOVE.L #Fenster,D1        ; Fensterdefinition nach D1
MOVE.L #Mode_Newfile,D2   ; Modus nach D2
JSR Open(A6)              ; Interne Unterroutine
RTS

; **** Warteschleife ****
Warten:
MOVE.L Dosbase,A6         ; Basisadresse nach A6
MOVE.L #100,D1            ; Wartezeit nach D1
JSR Delay(A6)             ; Warteroutine 1/50 Sek.
RTS

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Dosbase,A6         ; Dos-Basisadresse nach A6
MOVE.L Handle,D1          ; Fenster-Handle nach D1
JSR Close(A6)             ; Interne Unterroutine
RTS

; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L Dosbase,A1         ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)      ; Interne Unterroutine
RTS

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Speicherreservierung ****

EVEN
Dosbase:
DC.L 0                    ; Platz für Dos-Basisadresse
```

```
Handle:
DC.L 0                        ; Platz für Fenster-Handle

Fenster:
DC.B `CON:10/10/600/180/FENSTER`,0 ; Fensterdefinition

Dosname:
DC.B `dos.library`,0          ; Name der Dos.library
```

Speichern Sie das Programm nach der Eingabe unter dem Namen "Dos-Window" ab, und assemblieren Sie es. Wenn keine Fehler gemeldet werden, können Sie es wie gewohnt starten und den Erfolg begutachten.

Wie funktioniert das Programm?

Beginnen wir bei der Erklärung des Programms mit den bekannten Elementen des Hauptprogramms:

Zuerst benutzen wir das Unterprogramm OpenDos zum Öffnen der Dos.library. Im Hauptprogramm wird dann geprüft, ob alles geklappt hat und falls nicht zum Programmende verzweigt. Ist dagegen alles gutgegangen, merken wir uns die Basisadresse der Dos.library in der Speicherstelle Dosbase. Bekannt sind Ihnen sicherlich auch noch die beiden vom Hauptprogramm verwendeten Unterprogramme Warten und CloseDos. Beide wurden ja schon im letzten Kapitel erstellt und erläutert. Neu und interessant sind hingegen die beiden Unterprogramme OpenWin und CloseWin.

Das Unterprogramm OpenWin

Das Unterprogramm OpenWin benutzt das Unterprogramm Open der Dos.library, um das Fenster auf dem Bildschirm zu erstellen. Daher muß zuerst einmal die Basisadresse der Dos.library ins Adreßregister A6 geholt werden. Das Unterprogramm Open verlangt einen Dateinamen in D1 und einen speziellen Modus in D2.

Der Dateiname ist gerade unsere Fensterbeschreibung "CON:10/10/600/180/FENSTER", wie wir sie auch schon bei unseren Beispielen vom CLI aus verwendet haben. Wie bei anderen Texten

(z.B. dem Namen der Dos.library) muß der Text mit einer Null abgeschlossen werden, damit das Ende vom Betriebssystem erkannt werden kann.

Auf den Modus wollen wir an dieser Stelle nur ganz kurz eingehen, er spielt beim Öffnen von Dateien auf einer Diskette eine größere Rolle. Der Modus kann 1005 (Alt) oder 1006 (Neu) lauten, je nachdem, ob wir eine bestehende Datei (Alt) oder eine neue Datei (Neu) öffnen wollen. Für unser Fenster unterscheidet DOS da aber nicht, und so verwenden wir 1006.

Nachdem die Vorbereitungen getroffen wurden, wird die Unterroutine Open der Dos.library aufgerufen und zum Hauptprogramm zurückgekehrt. Das Hauptprogramm prüft nun den Rückgabewert in D0. Ist dieser Null, konnte DOS das Fenster nicht öffnen, und das Programm wird mit

```
BEQ CloseDos
```

beendet. Dies ist für spätere Erweiterungen sehr wichtig, da mit einem nicht geöffneten Fenster auf keinen Fall gearbeitet werden kann. Hat aber alles geklappt, wird der Rückgabewert in der Variablen Handle abgelegt. Dieser Rückgabewert ist ebenfalls für spätere Erweiterungen sehr wichtig - wir gehen dann ausführlicher darauf ein. Hier benötigen wir ihn nur für die folgende Routine CloseWin.

Das Unterprogramm CloseWin

Das Unterprogramm CloseWin verwendet zum Schließen des Windows eine weitere Unterroutine der Dos.library: Close. Folglich wird also wieder (sicherheitshalber) die Basisadresse der Dos.library ins Adreßregister A6 geladen. Damit DOS weiß, was denn nun geschlossen werden soll, wird die vorher von Open zurückgelieferte Zahl aus Handle nach D1 geholt. Dann kann Close aufgerufen und zum Hauptprogramm zurückgekehrt werden.

Halten wir also fest: Um ein DOS-Fenster zu öffnen, verwenden wir die Unterroutine Open der Dos.library und übergeben ihr

dieselbe Fensterbeschreibung, wie sie auch im CLI verwendet werden kann. Zurück bekommen wir eine spezielle Zahl, ein sogenanntes Handle. Diese Zahl müssen wir DOS immer übergeben, wenn wir etwas mit diesem Fenster machen wollen. Dies gilt natürlich auch für die Unterroutine Close aus der Dos.library, die unser Fenster nach zwei Sekunden wieder schließen soll. Das ist im CLI etwas anders: Dort schließt sich ein Ausgabe-Fenster, wenn nichts mehr auszugeben ist, und ein Eingabe-Fenster, wenn man <Ctrl> + <\> eingibt.

6.2 Die ersten Bildschirmausgaben - WRITE

Nachdem wir nun ein Fenster auf dem Bildschirm geöffnet haben, wollen wir natürlich auch Text hineinschreiben können. Auch das ist relativ einfach. Dazu stellt die Dos.library die Unterroutine Write zur Verfügung.

Das folgende Programm ist eine Erweiterung des vorher erstellten Basisprogramms Windows. Fügen Sie also hinter "Start:" die Zeile "Write = ...", die neue Zeile "JSR Schreiben" im Hauptprogramm, das Unterprogramm Schreiben und am Ende des Programms die Zeilen ab dem Label "Text:" ein. Das vollständige Programm sollte dann folgendermaßen aussehen:

```
Start:

; **** Variablen definieren ****
Execbase = 4                ; Adresse der Exec.library
OpenLibrary = -$0228        ; Offset
CloseLibrary = -$019E       ; Offset
Open = -$001E               ; Offset
Close = -$0024              ; Offset
Write = -$0030              ; Offset
Delay = -$00C6              ; Offset zum Warten

; **** Hauptprogramm ****
JSR OpenDos                 ; Dos.library öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ Ende                    ; Nein, Ende
MOVE.L D0,Dosbase           ; sonst: Adresse merken
JSR OpenWin                 ; Fenster öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ CloseDos                ; Nein, Dos.library schließen
```

```
MOVE.L D0,Handle          ; sonst: Adresse merken
JSR Schreiben
JSR Warten
JSR CloseWin
JMP CloseDos

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L #Dosname,A1        ; Libraryname nach 1
MOVE.L #0,D0              ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)       ; Interne Unterroutine
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Dosbase,A6         ; Dos-Basisadresse nach A6
MOVE.L #Fenster,D1        ; Fensterdefinition nach D1
MOVE.L #1006,D2           ; Access-Mode nach D2
JSR Open(A6)              ; Interne Unterroutine
RTS

; **** Texte schreiben ****
Schreiben:
MOVE.L Dosbase,A6         ; Dos-Basisadresse nach A6
MOVE.L Handle,D1          ; Fenster-Handle nach D1
MOVE.L #Text,D2           ; Textanfang nach D2
MOVE.L #Textende,D3       ; Textende nach D3
SUB.L D2,D3               ; Bestimmung der Länge
JSR Write(A6)             ; Interne Unterroutine
RTS

; **** Warteschleife ****
Warten:
MOVE.L Dosbase,A6         ; Basisadresse nach A6
MOVE.L #100,D1            ; Wartezeit nach D1
JSR Delay(A6)             ; Warteroutine 1/50 Sek.
RTS

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Handle,D1          ; Fenster-Handle nach D1
MOVE.L Dosbase,A6         ; Dos-Basisadresse nach A6
JSR Close(A6)             ; Interne Unterroutine
RTS

; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L Dosbase,A1         ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)      ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
```

```
ILLEGAL

; **** Speicherreservierung ****
EVEN
Dosbase:
DC.L 0                      ; Platz für Dos-Basisadresse

Handle:
DC.L 0                      ; Platz für Fenster-Handle

Dosname:
DC.B `dos.library`,0        ; Name der Dos.library

Fenster:
DC.B `CON:10/10/600/180/FENSTER`,0 ; Fensterdefinition

Text:
DC.B `Erster Text im Fenster`,10 ; Ausgabetext

Textende:                   ; Ende des Speichers
```

Speichern Sie das Programm am besten unter dem Namen "FensterAusgabe" ab, bevor Sie es assemblieren und starten.

Das Unterprogramm Schreiben

Kommen wir nun zur Erklärung der Funktionsweise des neuen Unterprogramms Schreiben. Da die Unterroutine Write in der Dos.library vorhanden ist, laden wir A6 sicherheitshalber mit der Basisadresse. Die Routine Write erwartet nun drei Angaben in den Registern D1-D3:

D1 Eine Information darüber, wohin die Ausgabe erfolgen soll. Das entsprechende Gerät oder die Datei muß vorher mit Open geöffnet worden sein. Das dabei erhaltene Handle muß man nun hier angeben. In unserem Fall verwenden wir gerade den Rückgabewert, den wir beim Öffnen des Fensters erhalten haben.

D2 Die Adresse des Textes, der ausgegeben werden soll.

D3 Die Länge des Textes in Bytes.

Während das Handle und die Adresse des Textes leicht anzugeben sind, ist das bei der Textlänge etwas problematischer. Natürlich hätten wir einfach die Anzahl der Zeichen auszählen und

in D3 schreiben können. Das hätte aber den entscheidenden Nachteil, daß wir keine Änderungen am Text vornehmen könnten.

Daher benutzen wir einfach einen Trick: Wir schreiben direkt hinter das letzte Zeichen des Textes ein weiteres Label "Textende". Damit haben wir also die Adresse des ersten Bytes hinter dem Text. Von diesem ziehen wir die Startadresse des Textes ab und erhalten so sofort im richtigen Register D3 die Länge. Nun kann der Text mit dem Aufruf der Unterroutine Write ausgegeben werden.

Hinweis: Der Text endet übrigens mit einer 10, weil dadurch der Cursor in eine neue Zeile gesetzt wird (Linefeed).

6.3 Die ersten Tastatureingaben - READ

Kommen wir nun zum Gegenstück des Schreibens: Wie kann man Eingaben von der Tastatur holen? Dazu gibt es eine entsprechende Unterroutine READ in der Dos.library. Während Write einen Text aus dem Speicher des Amiga ausgibt, holt Read Eingaben von der Tastatur in den Speicher.

Das folgende Programm ist eine Erweiterung des vorherigen Programms. Neben der Definition von Read zu Beginn des Programms wurde ein entsprechender Aufruf im Hauptprogramm und das Unterprogramm Lesen eingefügt:

```
Start:

; **** Variablen definieren ****
Execbase = 4                 ; Adresse der Exec.library
OpenLibrary = -$0228         ; Offset
CloseLibrary = -$019E        ; Offset
Open = -$001E                ; Offset
Close = -$0024               ; Offset
Write = -$0030               ; Offset
Read = -$002A                ; Offset

; **** Hauptprogramm ****
```

```
JSR OpenDos                  ; Dos.library öffnen
CMP.L #0,D0                  ; Hat geklappt?
BEQ Ende                     ; Nein, Ende
MOVE.L D0,Dosbase            ; sonst: Adresse merken
JSR OpenWin                  ; Fenster öffnen
CMP.L #0,D0                  ; Hat geklappt?
BEQ CloseDos                 ; Nein, Dos.library schließen
MOVE.L D0,Handle             ; sonst: Adresse merken
JSR Schreiben
JSR Lesen
JSR CloseWin
JMP CloseDos

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6           ; Basisadresse nach A6
MOVE.L #Dosname,A1           ; Libraryname nach 1
MOVE.L #0,D0                 ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)          ; Interne Unterroutine
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
MOVE.L #Fenster,D1           ; Fensterdefinition nach D1
MOVE.L #1006,D2              ; Access-Mode nach D2
JSR Open(A6)                 ; Interne Unterroutine
RTS

; **** Text schreiben ****
Schreiben:
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
MOVE.L Handle,D1             ; Fenster-Handle nach D1
MOVE.L #Text,D2              ; Speicher nach D2
MOVE.L #Textende,D3          ; Speicherende nach D3
SUB.L D2,D3                  ; Bestimmung der Länge
JSR Write(A6)                ; Interne Unterroutine
RTS

; **** Text von Tastatur lesen ****
Lesen:
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
MOVE.L Handle,D1             ; Fenster-Handle nach D1
MOVE.L #Text,D2              ; Speicher nach D2
MOVE.L #Textende,D3          ; Speicherende nach D3
SUB.L D2,D3                  ; Bestimmung der Länge
JSR Read(A6)                 ; Interne Unterroutine
RTS

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Handle,D1             ; Fenster-Handle nach D1
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
JSR Close(A6)                ; Interne Unterroutine
```

```
RTS

; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L Dosbase,A1          ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)       ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Speicherreservierung ****
EVEN
Handle:
DC.L 0                     ; Platz für Fenster-Handle

Dosbase:
DC.L 0                     ; Platz für Dos-Basisadresse

Dosname:
DC.B `dos.library`,0       ; Name der Dos.library

Fenster:
DC.B `CON:10/10/600/180/FENSTER`,0 ; Fensterdefinition

Text:
DC.B `Erster Text im Fenster`   ; Ausgabetext

Textende:                  ; Ende des Speichers
```

Wenn Sie das Programm eingegeben und fehlerfrei assembliert haben, sollten Sie es vor dem Starten unter dem Namen "FensterEingabe" abspeichern.

Nach dem Start erscheint das Fenster mit dem bekannten Text und anschließend können Sie so lange Zeichen eingeben, bis Sie die Taste <Return> betätigen. Dadurch werden die Fenster wieder geschlossen und das Programm beendet.

Wenn Sie sich nun fragen, wo denn der eingegebene Text im Speicher vorhanden ist: Er wurden in demselben Speicherbereich abgelegt, wo auch der vorher ausgegebene Text war. Wenn Sie sich die Eingaben anschauen wollen, können Sie im Befehlsmodus des SEKA einfach eingeben:

```
Q Text
```

Anschließend zeigt der SEKA den Speicher ab der Adresse "Text".

Das Unterprogramm Lesen

Kommen wir nun zur Erklärung der Funktionsweise des neuen Unterprogramms Lesen. Da die Unterroutine Read in der Dos.library vorhanden ist, laden wir A6 mit der Basisadresse der Dos.library. Dies ist zwar in unserem Programm nicht nötig, weil die Basisadresse dort noch steht, aber darauf sollte man sich besser nicht verlassen. Schnell fügt man einmal ein neues Unterprogramm ein, das eine andere Adresse nach A6 holt und schon stürzt das Programm plötzlich ab, weil ein völlig falscher Wert in A6 steht. Also: Besser in jeder Unterroutine alle Voraussetzungen schaffen und sich nicht auf vorherige Unterroutinen verlassen. Die Routine Read erwartet nun drei Angaben in den Registern D1-D3:

D1 Eine Information darüber, woher die Eingabe erfolgen soll. Das entsprechende Gerät oder die Datei muß vorher mit Open geöffnet worden sein. Das dabei erhaltene Handle muß man nun hier angeben. In unserem Fall verwenden wir gerade den Rückgabewert, den wir beim Öffnen des Fensters erhalten haben, denn dieses Fenster verwenden wir auch zur Eingabe.

D2 Die Adresse, ab der die gelesenen Zeichen im Speicher abgelegt werden sollen. Wir verwenden die Adresse des Textes, der ausgegeben werden soll.

D3 Die Länge des Textes in Bytes. Wir bestimmen die Länge des möglichen Textes wieder durch ein Label "Textende".

Im Unterprogramm Lesen werden nun die passenden Werte in die Register D1 und D2 geladen und die Länge für D3 berechnet. Dann kann die Unterroutine Read aufgerufen werden. Diese wird erst beendet, wenn auf der Tastatur die Taste <Return> betätigt wird.

Hinweis: Werden mehr Zeichen über die Tastatur eingegeben, als in D3 angegeben war, dann schreibt die Unterroutine Read nur die erlaubten Zeichen in den Puffer. Das ist sehr wichtig, denn ansonsten würde über das Ende des vorhandenen Puffers geschrieben, und das kann zur Zerstörung des Programms und damit zum Absturz führen.

Übrigens liefert uns das Betriebssystem im Register D0 zurück, wie viele Zeichen von der Tastatur geholt und im angegebenen Puffer abgelegt wurden. Das Zeichen mit dem Wert 10 (Linefeed/Return-Taste) zählt dabei übrigens mit.

6.4 Wie kommt man an das aktuelle CLI-Fenster?

Gut, Fenster öffnen, hineinschreiben und Tastatureingaben speichern können wir jetzt. Allerdings muß dazu immer erst ein spezielles Fenster erstellt werden. Für viele Zwecke wäre es aber viel interessanter, Informationen im aktuellen CLI-Fenster auszugeben. Dadurch würde beispielsweise der Nachteil entfallen, daß unsere Fenster entweder nach einer festgelegten Zeit oder erst nach Betätigung der Taste <Return> verschwinden.

Stellen Sie sich etwa vor, Ihr Programm ermittelt den freien Speicher und soll ihn ausgeben. Dazu wäre ein eigenes Fenster doch unnütz. Auch der CLI-Befehl AVAIL gibt ja beispielsweise die Informationen auf dem aktuellen CLI-Fenster aus.

Das spezielle Fenster "*"

Wenn Sie sich in Amiga-DOS ein wenig auskennen und Ihnen beispielsweise das Öffnen von Fenstern mit CON:10/10/300/100/Neues-Fenster bekannt war, dann kennen Sie vielleicht auch schon die Möglichkeit, an das aktuelle CLI-Fenster heranzukommen. Dazu braucht man nämlich nur als Dateinamen das Sternchen "*" zu verwenden.

Die folgende Befehlszeile in Amiga-DOS kopiert beispielsweise den Inhalt der Startup-Sequence in das aktuelle CLI-Fenster und entspricht damit in etwa dem Befehl TYPE:

```
COPY S:Startup-Sequence *
```

Das funktioniert genauso in unserem Programm. Ersetzen Sie also am Ende des Programms die Zeile:

```
DC.B `CON:10/10/600/180/FENSTER`,0        ; Fensterdefinition
```

durch folgende neue Zeile:

```
DC.B `*`,0                ; CLI-Fenster
```

Nach dem Assemblieren können Sie es starten. Tatsächlich wird der Text direkt hinter der Eingabeaufforderung (Prompt) des CLI ausgegeben. Anschließend wartet das Programm, bis Sie im CLI (eventuell nach Eingabe von Text) die Taste <Return> betätigt haben. Dann endet das Programm wie üblich.

Damit haben wir also einen einfachen und interessanten Weg gefunden, Informationen im aktuellen CLI-Fenster auszugeben. Damit lassen sich auch praktisch schon die ersten, sinnvollen Programme schreiben.

Hinweis: Die Eingabe von Texten aus dem aktuellen CLI-Fenster funktioniert nach unserer Erfahrung aber nicht ganz korrekt, wenn die Programme im SEKA gestartet werden. Am besten benutzen Sie innerhalb des SEKA für Texteingaben ein CON-Fenster. Im folgenden Kapitel zeigen wir Ihnen, wie Sie Programme auch außerhalb des SEKA starten können. Für diese Programme funktionieren Ein-/Ausgaben über das CLI-Fenster tadellos, und Sie sollten diese Möglichkeit auch verwenden.

6.5 Programme ohne SEKA starten

Da wir nun schon von unserem Programm aus mit der Außenwelt in Verbindung treten können, kommt langsam die Zeit für ein paar Hinweise, wie selbstgeschriebene Programme auch außerhalb des SEKA gestartet werden können. Sie wollen ja sicherlich kein Programm schreiben, daß den aktuellen Speicherplatz ermittelt, aber nur im SEKA gestartet werden kann.

Um ein Programm ohne den SEKA starten zu können, sind nur zwei Schritte nötig:

1. Bisher haben wir am Ende des Programms mit dem speziellen Befehl ILLEGAL die Kontrolle an den SEKA übergeben. Da nun kein SEKA mehr vorhanden ist, würde der Befehl ILLEGAL zu einem Programmabsturz führen. Statt dessen müssen wir den Befehl RTS verwenden. Vielleicht fragen Sie sich jetzt, warum denn RTS, denn dies ist doch der Rücksprungbefehl aus einem Unterprogramm. Tatsächlich ruft das Betriebssystem des Amiga unseres wie eins seiner Unterprogramme auf, und somit müssen wir unser Programm mit RTS beenden.

2. Bisher konnten wir nur Programmtexte (Source) speichern und laden. Wir müssen nun dem SEKA mitteilen, daß er unser Programm als lauffähiges Maschinenspracheprogramm abspeichern soll. Dazu dient ein spezieller Befehl:

   ```
   WO
   ```

   Diese Abkürzung steht für Write Object und fragt anschließend nach dem Namen für das Programm.

Hinweis: Bisher konnten wir noch immer im Schutz des SEKA ausprobieren. Das hatte zwei Vorteile: Zum einen brauchten wir uns um das Vorher und Nachher nicht zu kümmern, zum anderen nahm der Amiga uns kleinere Programmierfehler nicht unbedingt übel. Sollen Programme auch außerhalb des SEKA funk-

tionieren, dürfen Sie keine Fehler enthalten. Während der SEKA meistens die Guru Meditation (also den Absturz des Amiga) verhindert und nur einen Fehler meldet, führen solche Programmfehler außerhalb des SEKA schnell zum Absturz.

Um unser Programm "FensterEingabe" ohne SEKA starten zu können, entfernen wir hinter dem Label "Ende:" den Befehl ILLEGAL und schreiben dorthin RTS. Anschließend muß das Programm natürlich mit A neu assembliert werden. Zum Abspeichern verwenden Sie den Befehl

```
WO
```

und geben als Namen ebenfalls "FensterEingabe" an. Sie brauchen dabei keine Angst zu haben, die gleichnamige Datei mit dem Quelltext des Programms zu überschreiben. Diese bekommt vom SEKA beim Speichern automatisch die Endung ".S", das mit WO gespeicherte, lauffähige Programm nicht.

Anschließend können Sie ein neues CLI öffnen und das Programm "FensterEingabe" direkt durch Angabe des Namens starten. Alles verläuft wie gewohnt: Der Text wird ausgegeben, und nach <Return> endet das Programm.

Hinweis: Übrigens ist der Wert, der beim Programmende in D0 steht und an das Betriebssystem zurückgeliefert wird, eine Rückmeldung, wie wir sie auch in eigenen Unterprogrammen häufig verwendet haben, und kann beispielsweise in Batch-Dateien zum Abbruch der Batch-Datei verwendet werden. Wenn Sie keinen Fehler an das Betriebssystem zurückliefern wollen, sollten Sie eine Null hineinschreiben, bevor das Programm endet.

7. Der Zugriff auf Disketten

In diesem Abschnitt wollen wir Ihnen zeigen, wie man von eigenen Programmen auf Dateien von Disketten zugreifen kann.

7.1 Dateien öffnen und lesen

Eigentlich haben wir für das Bearbeiten von Dateien auf einer Diskette schon im letzten Kapitel alles Wesentliche kennengelernt. Dort haben wir nämlich mit Amiga-DOS ein Fenster geöffnet und in dieses Fenster Texte geschrieben und Eingaben von der Tastatur geholt.

Das Besondere an Amiga-DOS ist, daß dieser Vorgang für Dateien völlig identisch ist, es werden auch dieselben Unterroutinen der Dos.library zum Öffnen, Lesen, Schreiben und Schließen verwendet. Einige kleine Unterschiede gibt es aber doch, so daß dieses Kapitel des Buches nicht völlig überflüssig ist. So kann beim Öffnen einer Datei diese beispielsweise gar nicht vorhanden sein, oder beim Schreiben in die Datei kann ein Schreibschutz zu einem Fehler führen.

Programm: Startup-Sequence öffnen und lesen

Beginnen wir zuerst mit einem recht einfachen Programm, daß viele schon im letzten Kapitel erstellte Unterroutinen benutzt: Das folgende Programm öffnet die Datei S:Startup-Sequence und gibt diese auf dem Bildschirm im aktuellen CLI-Fenster aus.

```
Start:

; **** Variablen definieren ****
Execbase = 4                ; Adresse der Exec.library
Mode_Newfile = 1006         ; Datei-Modus: neue Datei
Mode_Oldfile = 1005         ; Datei-Modus: bestehende Datei
OpenLibrary = -$0228        ; Offset
CloseLibrary = -$019E       ; Offset
Open = -$001E               ; Offset
```

```
Close = -$0024              ; Offset
Write = -$0030              ; Offset
Read = -$002A               ; Offset
AnzPuffer = 256             ; Puffergröße zum Lesen

; **** Hauptprogramm ****
JSR OpenDos                 ; Dos.library öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ Ende                    ; Nein, Ende
MOVE.L D0,Dosbase           ; sonst: Adresse merken
JSR OpenWin                 ; Fenster öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ CloseDos                ; Nein, Dos.library schließen
MOVE.L D0,WinHandle         ; sonst: Adresse merken
JSR OpenFile                ; Datei öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ CloseWinDos             ; Nein, Window+Library zu
MOVE.L D0,DateiHandle       ; sonst: Adresse merken
JSR Schreiben               ; Infotext ausgeben
JSR Dateiausgeben           ; Datei im Fenster ausgeben
JSR CloseWin
JMP CloseDos

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L #Dosname,A1          ; Libraryname nach 1
MOVE.L #0,D0                ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)         ; Interne Unterroutine
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Dosbase,A6           ; Dos-Basisadresse nach A6
MOVE.L #Fenster,D1          ; Fensterdefinition nach D1
MOVE.L #Mode_Newfile,D2     ; Access-Mode nach D2
JSR Open(A6)                ; Interne Unterroutine
RTS

; **** Öffnen der Datei ****
OpenFile:
MOVE.L Dosbase,A6           ; Dos-Basisadresse nach A6
MOVE.L #Datei,D1            ; Fensterdefinition nach D1
MOVE.L #Mode_Oldfile,D2     ; Access-Mode nach D2
JSR Open(A6)                ; Interne Unterroutine
RTS

; **** Text schreiben ****
Schreiben:
MOVE.L Dosbase,A6           ; Dos-Basisadresse nach A6
MOVE.L WinHandle,D1         ; Fenster-Handle nach D1
MOVE.L #Text,D2             ; Speicher nach D2
MOVE.L #Textende,D3         ; Speicherende nach D3
SUB.L D2,D3                 ; Bestimmung der Länge
```

```
JSR Write(A6)              ; Interne Unterroutine
RTS

; **** Datei lesen und ausgeben ****
Dateiausgeben:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L Dateihandle,D1      ; Datei-Handle nach D1
MOVE.L #Puffer,D2          ; Pufferadresse nach D2
MOVE.L #Anzpuffer,D3       ; Anzahl zu lesender Zeichen
JSR Read(A6)               ;
MOVE.L D0,Anzgelesen       ; Anzahl merken
CMP.L #0,D0                ; Gab es noch ein Zeichen?
BEQ DateiausgebenEnde      ; keine Zeichen mehr, Ende
MOVE.L Winhandle,D1        ; Fenster-Handle nach D1
MOVE.L #Puffer,D2          ; Pufferadresse nach D2
MOVE.L Anzgelesen,D3       ; Anzahl gelesener Zeichen
JSR Write(A6)
CMP.L #Anzpuffer,Anzgelesen ; waren weniger Zeichen da ?
BEQ Dateiausgeben          ; nein, mehr Zeichen ausgeben

DateiausgebenEnde:
RTS

; **** Schließen des Fensters ****
CloseWin:
MOVE.L WinHandle,D1        ; Fenster-Handle nach D1
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
JSR Close(A6)              ; Interne Unterroutine
RTS

; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L Dosbase,A1          ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)       ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Window und Dos bei Fehler schließen ****
CloseWinDos:
JSR CloseWin
JMP CloseDos

; **** Speicherreservierung ****
EVEN
Dosbase:
DC.L 0                     ; Platz für Dos-Basisadresse

WinHandle:
DC.L 0                     ; Platz für Fenster-Handle

Dateihandle:
```

```
DC.L 0                     ; Platz für Datei-Handle

Anzgelesen:
DC.L 0                     ; Variable für gelesene Zeichen

Dosname:
DC.B `dos.library`,0       ; Name der Dos.library

Fenster:
DC.B `*`,0                 ; Fensterdefinition

Datei:
DC.B `S:Startup-Sequence`,0 ; Name der Datei

Text:
DC.B `Inhalt der Startup-Sequence`,10 ; Meldungstext

Textende:                  ; Ende des Speichers

Puffer:
BLK.B AnzPuffer,0
```

Speichern Sie das Programm beispielsweise unter dem Namen Readfile ab, und assemblieren Sie es. Nach dem Starten wird die Startup-Sequence aus dem Verzeichnis S: im aktuellen CLI-Fenster ausgegeben.

Wie funktioniert das Programm?

Das Programm enthält natürlich wieder viele Elemente der vorherigen Programme. Neu hinzugekommen ist vor allem die Definition der Puffergröße "Anzpuffer" zu Beginn des Programms. Wir hätten natürlich auch an den beiden Stellen im Programm, wo diese Größe verwendet wird, direkt 256 verwenden können, aber wenn man diese Zahl einmal ändert, vergißt man leicht die andere Stelle zu ändern, und das Programm stürzt ab.

Im Hauptprogramm sind zwei neue Unterprogrammaufrufe hinzugekommen: JSR OpenFile und JSR Dateiausgeben. Das Unterprogramm OpenFile arbeitet genauso wie die Unterroutine zum Öffnen des CLI-Fensters, nur wird der Dateiname der Startup-Sequence angegeben. Zusätzlich ist wichtig, daß wir als Modus beim Öffnen der Datei 1005 (Alt) angeben. Mit diesem Modus wird nämlich eine bereits bestehende Datei geöffnet. Geben wir

statt dessen 1006 (Neu) an, wird die Datei neu erstellt, und das wollen wir bei der vorhandenen Datei Startup-Sequence natürlich nicht.

Das Unterprogramm Dateiausgeben

Interessanter ist da schon das neue Unterprogramm "Dateiausgeben". Es holt Zeichen aus der Datei und gibt sie auf dem Bildschirm im aktuellen CLI-Fenster aus. Ganz einfach beschrieben besteht es aus zwei Komponenten:

1. Eine bestimmte Anzahl Zeichen aus der Datei in einen Puffer lesen.

2. Anschließend den Puffer im CLI-Fenster ausgeben.

Ganz so einfach ist das Problem aber nicht zu lösen, denn wir müssen für einige Fälle Vorkehrungen treffen:

- Wenn die ersten 256 Zeichen gelesen und ausgegeben wurden, muß das Unterprogramm wieder von vorne beginnen.

- Wenn keine Zeichen mehr gelesen werden konnten, dürfen auch keine Zeichen ausgegeben werden, und das Unterprogramm muß enden. Die Anzahl der gelesenen Zeichen erhalten wir übrigens nach dem Aufruf von Read in D0 zurück.

- Ein spezieller Fall tritt immer beim Ende der zu lesenden Datei auf: Dann werden nämlich nicht mehr 256 Bytes aus der Datei gelesen, sondern weniger. Dann dürfen natürlich auch nur weniger Zeichen im Fenster ausgegeben werden und außerdem sollte unser Unterprogramm dann ebenfalls - aber diesmal nach der Ausgabe der Daten - enden.

All dies erledigt das Unterprogramm "Dateiausgeben:". Wie üblich wird sicherheitshalber die Basisadresse der Dos.library in das Adreßregister A6 geholt. Dann werden die Register D1-D3 mit den von Read erwarteten Werten gesetzt. Dabei versuchen wir, 256 Bytes also einen ganzen Puffer mit Daten zu füllen.

Nach dem Aufruf von Read merken wir uns die Anzahl gelesener Bytes für spätere Verwendung in der Variablen Anzgelesen.

Anschließend muß natürlich geprüft werden, ob überhaupt noch mindestens ein Byte gelesen wurde. Ist dies nicht der Fall, wird natürlich auch nichts ausgegeben und zum Ende des Unterprogramms verzweigt.

Konnten aber Zeichen gelesen werden, werden die Register mit den Werten für die Unterroutine Write gesetzt. Dabei wird als Anzahl zu schreibender Zeichen genau die Anzahl gelesener Zeichen verwendet. Nach dem Aufruf von Read prüft das Unterprogramm, ob überhaupt noch Daten für den gesamten Puffer gelesen werden konnten (CMP.L #Anzpuffer,Anzgelesen). Ist dies der Fall, sind wahrscheinlich auch weitere Daten vorhanden, und es wird zum Anfang des Unterprogramms verzweigt, ansonsten zum Hauptprogramm zurückgekehrt.

Hinweis: Inzwischen werden von unserem Hauptprogramm schon einige Unterprogramme aufgerufen, und in den meisten Fällen wird geprüft, ob denn auch alles im Unterprogramm geklappt hat. Was aber, wenn etwas nicht geklappt hat. Bei unseren ersten Programmen konnten wir noch ganz einfach zum Programmende springen. Inzwischen ist das aber nicht mehr möglich (oder erlaubt), denn wenn wir beispielsweise ein Fenster geöffnet haben, aber die Datei nicht finden können, dürfen wir nicht einfach aufhören, sondern müssen das Fenster wieder ordnungsgemäß schließen.

In unserem Programm haben wir eine spezielle Routine CloseWinDos geschrieben, die das Fenster und anschließend die Dos.library schließt. In noch komplexeren Programmen werden wir uns etwas Besseres einfallen lassen.

7.2 Neue Dateien erstellen und schreiben

So wie wir bereits bestehende Dateien öffnen und aus ihnen lesen können, können wir auch neue Dateien erstellen und in diese etwas hineinschreiben. Wenn man eine bestehende Datei öffnet, alle Daten aus ihr liest und in eine neue Datei schreibt, hat man praktisch einen DOS-Befehl COPY. Da dieser Befehl aber schon besteht, haben wir uns als Beispiel etwas einfallen lassen, was noch nicht auf Ihrer Workbench-Diskette vorhanden ist und was Sie vielleicht gut gebrauchen können.

Das folgende Programm Codefile verschlüsselt eine Datei. Anschließend kann man eine Textdatei also nicht mehr lesen oder eine Programmdatei nicht mehr ausführen. Wendet man unser Programm aber noch einmal auf dieselbe Datei an, ist sie wieder normal lesbar oder ausführbar.

```
Start:

; **** Variablen definieren ****
Execbase = 4                  ; Adresse der Exec.library
Mode_Newfile = 1006           ; Datei-Modus: neue Datei
Mode_Oldfile = 1005           ; Datei-Modus: bestehende Datei
OpenLibrary = -$0228          ; Offset
CloseLibrary = -$019E         ; Offset
Open = -$001E                 ; Offset
Close = -$0024                ; Offset
Write = -$0030                ; Offset
Read = -$002A                 ; Offset

; **** Hauptprogramm ****
JSR OpenDos                   ; Dos.library öffnen
CMP.L #0,D0                   ; Hat geklappt?
BEQ Ende                      ; Nein, Ende
MOVE.L D0,Dosbase             ; sonst: Adresse merken
JSR OpenWin                   ; Fenster öffnen
CMP.L #0,D0                   ; Hat geklappt?
BEQ CloseDos                  ; Nein, Dos.library schließen
MOVE.L D0,Handle              ; sonst: Adresse merken
JSR Schreiben                 ; Eingabeaufforderung ausgeben
JSR Lesen                     ; Dateinamen holen
JSR OpenFile                  ; Gewünschte Datei öffnen
CMP.L #0,D0                   ; Hat geklappt?
BEQ CloseWin                  ; Nein, ab Window schließen
JSR OpenNewFile               ; Neue Datei öffnen
CMP.L #0,D0                   ; Hat geklappt?
BEQ CloseEin                  ; Nein, ab Window schließen
```

```
JSR Dateikodieren          ; Datei lesen, kodiert schreiben
JMP CloseAus               ; Alles schließen und Ende

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L #Dosname,A1         ; Libraryname nach 1
MOVE.L #0,D0               ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)        ; Interne Unterroutine
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L #Fenster,D1         ; Fensterdefinition nach D1
MOVE.L #Mode_Newfile,D2    ; Access-Mode nach D2
JSR Open(A6)               ; Interne Unterroutine
RTS

; **** Text schreiben ****
Schreiben:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L Handle,D1           ; Fenster-Handle nach D1
MOVE.L #Text,D2            ; Speicher nach D2
MOVE.L #Textende,D3        ; Speicherende nach D3
SUB.L D2,D3                ; Bestimmung der Länge
JSR Write(A6)              ; Interne Unterroutine
RTS

; **** Dateiname von Tastatur lesen ****
Lesen:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L Handle,D1           ; Fenster-Handle nach D1
MOVE.L #Dateiname,D2       ; Platz für Name nach D2
MOVE.L #96,D3              ; Max 96 Zeichen lesen
JSR Read(A6)               ; Interne Unterroutine
MOVE.B D0,Dateinamelaenge  ; Länge des Namens merken
RTS

; **** Öffnen der Quelldatei ****
OpenFile:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L #Dateiname,D1       ; Fensterdefinition nach D1
MOVE.L #0,D2               ; Register D2 auf 0 setzen
MOVE.B Dateinamelaenge,D2  ; Länge des Namens holen
ADD.L D1,D2                ; Startadresse hinzuzählen
SUB.L #1,D2                ; 1 Byte weniger->letzes Byte, CR
MOVE.L D2,A0               ; ins Adreßregister
MOVE.B #0,(A0)             ; dort 0 hineinschreiben, mit 0
                           ; abschließen
MOVE.L #Mode_Oldfile,D2    ; Access-Mode "Alt" nach D2
JSR Open(A6)               ; Interne Unterroutine
```

```
MOVE.L D0,HandleEin        ; Eingabehandle merken
RTS

; **** Neue Datei für kodierte Daten öffnen ****
OpenNewFile:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L #Dateiname,D1       ; Fensterdefinition nach D1
MOVE.L #0,D2               ; Register D2 auf 0 setzen
MOVE.B Dateinamelaenge,D2  ; Länge des Namens holen
ADD.L D1,D2                ; Startadresse hinzuzählen -> letztes
                           ; Zeichen+1
SUB.L #1,D2                ; 1 Byte weniger->letzes Byte
MOVE.L D2,A0               ; ins Adreßregister
MOVE.B #'.',(A0)+          ; an den Namen ".COD" anhängen
MOVE.B #'C',(A0)+          ;
MOVE.B #'O',(A0)+          ;
MOVE.B #'D',(A0)+
MOVE.B #0,(A0)             ; dann mit 0 abschließen
MOVE.L #Mode_Newfile,D2    ; Access-Mode "Neu" nach D2
JSR Open(A6)               ; Interne Unterroutine
MOVE.L D0,HandleAus        ; Ausgabehandle merken
RTS

; **** Daten lesen, kodieren und schreiben ****
Dateikodieren:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L HandleEin,D1        ; Datei-Handle nach D1
MOVE.L #Puffer,D2          ; Pufferadresse nach D2
MOVE.L #01,D3              ; Ein Zeichen lesen
JSR Read(A6)               ;
CMP.L #0,D0                ; Anzahl gelesener Zeichen
BEQ DateikodierenEnde      ; keine Zeichen mehr, Ende
EOR.B #$FF,Puffer          ; Bits im Puffer umkehren -> kodieren
MOVE.L HandleAus,D1        ; Handle Ausgabedatei  nach D1
MOVE.L #Puffer,D2          ; Pufferadresse nach D2
MOVE.L #01,D3              ; Ein Zeichen schreiben
JSR Write(A6)
CMP.L #01,D0               ; Ein Zeichen geschrieben ?
BEQ Dateikodieren          ; ja, alles klar, weiter
MOVE.L #$FFFFFFFF,D0       ; Rückgabewert: Fehler, nicht korrekt
                           ; geschrieben

DateikodierenEnde:
EOR.L #$FFFFFFFF,D0        ; Rückgabewert umdrehen
RTS

; **** Schließen der Ausgabedatei ****
CloseAus:
MOVE.L HandleAus,D1        ; Fenster-Handle nach D1
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
JSR Close(A6)              ; Interne Unterroutine
```

```
; **** Schließen der Eingabedatei ****
CloseEin:
MOVE.L HandleEin,D1          ; Eingabe-Handle nach D1
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
JSR Close(A6)                ; Interne Unterroutine

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Handle,D1             ; Fenster-Handle nach D1
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
JSR Close(A6)                ; Interne Unterroutine

; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6           ; Basisadresse nach A6
MOVE.L Dosbase,A1            ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)         ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Speicherreservierung ****

EVEN
Dosbase:
DC.L 0                       ; Platz für Dos-Basisadresse

Handle:
DC.L 0                       ; Platz für Fenster-Handle

HandleEin:
DC.L 0                       ; Handle für Eingabedatei

HandleAus:
DC.L 0                       ; Handle für Ausgabedatei

Dosname:
DC.B `dos.library`,0         ; Name der Dos.library

Fenster:
DC.B `CON:10/10/600/180/FENSTER`,0  ; Fensterdefinition

Text:
DC.B `Bitte Dateinamen eingeben: `  ; Ausgabetext

Textende:                    ; Ende des Speichers

Dateiname:
Blk.B 100,0

Dateinamelaenge:
DC.B 0
```

```
Puffer:
DC.B 0                    ; Platz für ein Byte zum Dekodieren
```

Nach dem fehlerfreien Assemblieren speichern Sie das Programm mit W unter dem Namen Codefile auf Diskette ab. Es enthält dann automatisch die SEKA-Erweiterung ".S". Dann können Sie es starten.

Nach dem Start erscheint ein Fenster, in dem der Dateiname einer bestehenden Datei angegeben werden kann, beispielsweise S:Startup-Sequence. Diese Datei wird anschließend kodiert und erhält die zusätzliche Erweiterung ".COD". Sie können ja nach dem Programmende einmal probieren, diese Datei mit TYPE anzuschauen - Sie werden ein "buntes" Wunder erleben. Um die Datei wieder lesbar zu machen, starten Sie das Programm erneut und geben den Namen der codierten Datei an, beispielsweise S:Startup-Sequence.COD. Am Ende des Programms haben Sie dann die wieder entschlüsselte Datei mit dem Namen S:Startup-Sequence.COD.COD. Sie können leicht mit TYPE nachprüfen, daß diese Datei wirklich exakt dem Inhalt der Startup-Sequence entspricht.

Das Unterprogramm OpenFile

Bevor dieses Unterprogramm aufgerufen wird, wurde im Fenster eine Eingabeaufforderung ausgegeben und der Dateiname eingegeben. Dabei wurde die Länge des eingegebenen Namens in "Dateinamelaenge" abgelegt. Das Unterprogramm OpenFile bestimmt nun das letzte Zeichen des Dateinamens, (normalerweise das abschließende Linefeed) und schreibt dort die von AmigaDOS erwartete 0 hinein. Dann wird die Datei geöffnet und das Handle abgelegt. Das Hauptprogramm prüft anschließend, ob die Datei korrekt geöffnet werden konnte und bricht gegebenenfalls ab.

Das Unterprogramm OpenNewFile

Das Unterprogramm OpenNewFile bestimmt das letzte Zeichen hinter dem Dateinamen (vor der Null) und fügt die Zeichen ".COD" an. Dabei wird jeweils ein Zeichen an die aktuelle Posi-

tion (in A0) geschrieben und mit demselben Befehl A0 ein Zeichen weitergesetzt. Dies macht das Pluszeichen hinter der Klammer. Der Befehl

```
MOVE.B #'.',(A0)+
```

bedeutet also: Schreibe das Zeichen "." an die Adresse, die im Adreßregister A0 steht und erhöhe das Adreßregister um ein Byte. Dieser Befehl kann auch mit der Operandenlänge ".W" und ".L" verwendet werden, dann wird A0 dementsprechend um zwei (Wort) oder vier Bytes (Langwort) erhöht.

Zum Schluß schließt das Unterprogramm den neuen Dateinamen mit einer Null ab und öffnet die Datei mit dem Modus "Neu". Dadurch wird diese von Amiga-DOS neu angelegt.

Das Unterprogramm Dateikodieren

Das Unterprogramm Dateikodieren arbeitet in drei Schritten:

1. Ein Zeichen aus der bestehenden Datei lesen; konnte kein Zeichen mehr gelesen werden, Unterprogramm beenden.

2. Das gelesene Zeichen durch das Umdrehen aller Bits kodieren. Dazu dient der Befehl EOR. Als erstes wird angegeben, welche Bits umgedreht werden sollen ($FF = %11111111 also alle Bits) und dann folgt das Ziel, in unserem Fall das Zeichen im Puffer. Das Interessante an diesem Befehl EOR ist, daß zweimalige Anwendung gerade wieder den ursprünglichen Zustand herstellt. Wenn wir eine Datei also kodiert haben und die kodierte Datei erneut verschlüsseln, erhalten wir wieder das ursprüngliche, lesbare Ergebnis.

3. Im dritten Schritt wird dann das kodierte Zeichen in die neue Datei geschrieben. Anschließend wird geprüft, ob wirklich ein Zeichen geschrieben werden konnte. Ist dies der Fall, wird das Unterprogramm erneut durchlaufen, ansonsten zur Fehlerroutine gesprungen.

Nutzung des Programms ohne SEKA

Da dies Programm nun schon recht nützlich ist, sollten Sie es so abändern, daß es auch ohne den SEKA von Amiga-DOS aus gestartet werden kann:

- Ändern Sie das ILLEGAL hinter dem Label "Ende:" in den Befehl RTS um.

- Ersetzen Sie die Zeile mit der Fensterdefiniton

  ```
  DC.B `CON:10/10/600/180/FENSTER`,0
  ```

  durch eine neue Zeile, die Ein- und Ausgaben über das CLI-Fenster ermöglicht:

  ```
  DC.B `*`,0
  ```

- Assemblieren Sie das Programm mit A neu, und speichern Sie anschließend das lauffähige Programm mit WO unter dem Namen Codefile ab. Dann können Sie es vom CLI aus wie einen Amiga-DOS-Befehl aufrufen und jederzeit Dateien verschlüsseln und entschlüsseln.

Hinweis: Wenn Sie das Programm zum Schutz von Texten und Programmen einsetzen wollen, können Sie sich eine "ganz persönliche" Verschlüsselung erstellen. Ändern Sie dazu im Unterprogramm "Dateikodieren" in der Zeile

```
EOR.B #$FF,Puffer
```

den Verschlüsselungswert $FF in einen anderen Wert um. Nehmen Sie nur keinen Wert, in dem zu wenige Bits = 1 sind, denn nur diese Bits werden umgedreht. Schreiben Sie die Zahl doch einfach statt in hexadezimaler in binärer Schreibweise hin, also beispielsweise:

```
EOR.B #%10100111,Puffer
```

Damit haben Sie ein ganz persönliches Verschlüsselungsprogramm, und jemand kann die Daten nur entschlüsseln, wenn er ebenfalls dieselbe Zahl verwendet. Da aber 255 Kombinationen möglich sind, wird das so schnell nicht vorkommen.

Außerdem sollten Sie bei der Nutzung des Programms am Ende des Hauptprogramms hinter "JSR Dateikodieren" noch den Rückgabewert überprüfen und gegebenenfalls (falls ein Fehler beim Kodieren aufgetreten ist) eine Fehlermeldung auf dem Bildschirm ausgeben. Wir haben darauf verzichtet, um das Programm nicht noch zu verlängern. Die Vorgehensweise finden Sie beispielsweise im Programm COMP zum Vergleichen von Dateien.

7.3 Einschub - So erhält man Zahlen von der Tastatur

Im nächsten Kapitel wollen wir ein weiteres, sehr nützliches Programm erstellen, mit dem beliebige Speicherbereiche aus dem Amiga auf Diskette geschrieben werden können. Damit Sie aber den Speicherbereich über die Tastatur angeben können, brauchen wir ein Unterprogramm, mit dem über die Tastatur eingegebene Ziffern in eine Zahl umgewandelt werden können. Dieses Unterprogramm wollen wir in diesem Kapitel erstellen.

Das folgende Programm wandelt eine acht Zeichen lange Ziffernfolge in hexadezimaler Schreibweise in eine Zahl (Langwort) im Register D2 um. Die Ziffernfolge muß sich dazu in einem Speicherbereich ab dem Label "Puffer:" befinden. Außerdem liefert das Programm in D0 den Wert 0, wenn bei der Umwandlung ein Fehler aufgetreten ist (beispielsweise weil ein ungültiges Zeichen eingegeben wurde) oder den Wert $FF, wenn das Umwandeln in eine Zahl problemlos geklappt hat.

```
GetZahl:
LEA Puffer,A0           ; Pufferadresse holen
CLR.L D0
CLR.L D2                ; Zahl = 0
```

```
MOVE.L #8,D1                    ; Zähler für Anzahl Ziffern
DoZiffer:
MOVE.B (A0)+,D0                 ; Ziffer holen
JSR Value                       ; in Wert umwandeln
CMP.B #$FF,D0                   ; Fehler ?
BEQ GetZahlFehler               ; War keine Ziffer -> Fehler und Ende
ADD.L D0,D2                     ; Ziffer zu Zahl addieren
SUB.L #1,D1                     ; Ziffern fertig, Ende
BEQ GetZahlOK                   ; Ziffern fertig, Ende
ASL.l #4,D2                     ; Zahl * 16
JMP DoZiffer                    ; und weiter

GetZahlFehler:
MOVE.L #0,D0
ILLEGAL

GetZahlOK:
MOVE.L #$FF,D0
ILLEGAL

Value: ; holt für eine Ziffer in D0 den Wert nach D0

; zuerst prüfen, ob Zeichen ein Kleinbuchstabe ist
CMP.B #'a',D0                   ; kleinster Kleinbuchstabe?
BLT Value_Gross                 ; Zeichen ist kleiner
CMP.B #'z',D0                   ; größter Kleinbuchstabe?
BGT Value_Gross                 ; Zeichen ist größer
; Das Zeichen ist also zwischen "a" und "f"...
CMP.B #'f',D0                   ; größer als "f"
BGT NoHex                       ; ja, keine Hex-Ziffer
SUB.B #'a',D0                   ; -> "a" = 0, "b" = 1 usw
ADD.B #10,D0                    ; -> "a" = 10, "b" = 2 usw.
RTS                             ; fertig

; jetzt prüfen, ob Zeichen ein Großbuchstabe ist
Value_Gross:
CMP.B #'A',D0
BLT Value_Ziffer                ; kleiner? Ziffer prüfen
CMP.B #'Z',D0
BGT Value_Ziffer                ; größer? Ziffer prüfen
CMP.B #'F',D0
BGT NoHex                       ; Nur "A"-"F" sind Hex-Ziffern
SUB.B #'A',D0
ADD.B #10,D0
RTS

; Zeichen ist kein Buchstabe, also auf Ziffer prüfen
Value_Ziffer:
CMP.B #'0',D0
BLT NoHex                       ; kleiner als Ziffer "0"
CMP.B #'9',D0
BGT NoHex                       ; größer als Ziffer "9"
```

```
SUB.B #'0',D0               ; "0"->0, "1"->1 usw.
RTS                         ; fertig

NoHex: MOVE.B #$FF,D0       ; Fehler, keine Hexzahl
RTS

Puffer: DC.B 'ABCD1234'
```

Speichern Sie das Programm nach der Eingabe unter dem Namen Getzahl ab. Nach dem Assemblieren können Sie es starten. Wenn Sie das Programm richtig eingegeben haben, erhalten Sie nach dem Programmende im Register D0 den Wert $FF (für: kein Fehler aufgetreten) und in D2 den korrekt umgewandelten Wert $ABCD1234.

Das Unterprogramm Value

Das Kernstück des Programms ist ein Unterprogramm Value, das genau eine Ziffer in eine Zahl umwandeln kann. Diese Ziffer wird als Zahl zwischen 0 und 15 in D0 zurückgeliefert, entsprechend dem möglichen Wert der Hexziffer $0 - $F. Dabei prüft das Unterprogramm Value, ob es sich um einen Kleinbuchstaben, einen Großbuchstaben oder eine Ziffer handelt, und dementsprechend erfolgt die Umwandlung in einem der drei Programmteile. Ist das Zeichen in D0 keine Ziffer zwischen "a"-"z", "A"-"Z" oder "0"-"9", wird in D0 der Fehlerwert $FF zurückgeliefert.

Hinweis: Vielleicht fragen Sie sich, warum unser Unterprogramm "Value" entgegen der sonstigen Vorgehensweise als Fehler $FF verwendet und nicht die übliche Null? Der Grund ist einfach: 0 ist ein möglicher, richtiger Rückgabewert wenn die Ziffer "0" bearbeitet wurde, während $FF nie auftreten kann und deshalb als Fehlererkennung verwendet wird.

Die Befehle BLT und BGT

Die Befehle sind dem Befehl BEQ sehr ähnlich. Sie gehören auch zur gleichen Kategorie von Befehlen und zwar zur Kategorie der Befehle, die bedingte Verzweigungen auslösen. BLT ist

die Abkürzung für BRANCH LOWER THAN, was übersetzt soviel bedeutet, wie: Verzweige, wenn kleiner als, während BGT für BRANCH GRATER THAN (also: verzweige wenn größer als) steht. Diese Befehle verzweigen also zum angegebenen Label, wenn die Bedingung erfüllt ist, ansonsten wird der nächste Befehl bearbeitet.

Hauptprogramm GetZahl

Das Hauptprogramm GetZahl holt zuerst die Adresse des Puffers mit der Ziffernfolge in das Register A0. Dort steht für unseren Test die Hexzahl $ABCD1234 als Ziffernfolge. Später wird hier der über die Tastatur eingegebene Inhalt stehen.

Mit LEA haben wir einen neuen Befehl verwendet, der immer dann verwendet werden kann, wenn eine Adresse in ein Adreßregister geholt werden soll. Die Abkürzung LEA steht deshalb auch für Load Effective Adress (Lade eine effektive Adresse). Er kann nur mit Adreßregistern verwendet werden und bewirkt in unserer Zeile genau dasselbe wie:

```
MOVE.L #Puffer,A0
```

Im Gegensatz zum Befehl MOVE kann man hier aber nicht das Zeichen "#" vergessen, wenn man die Adresse eines Labels verwenden will.

Anschließend werden die beiden Register D0 und D2 gelöscht. Dies ist notwendig, da das Unterprogramm diese Register nur byteweise bearbeitet und somit könnten "Zahlenreste" in den Registern vorhanden sein. Das Register D0 wird zur Umwandlung einer Ziffer in eine Zahl verwendet, in D2 wird in mehreren Durchläufen die komplette Zahl zusammengesetzt. Um die Anzahl der Durchläufe festzulegen, wird ein Zähler in D1 gesetzt (MOVE.L #8,D1). Zum Löschen der beiden Register haben wir einen neuen und speziellen Befehl verwendet: CLR. Mit diesem Befehl können Sie ein Datenregister oder eine Speicherstelle besonders einfach löschen. CLR.L D0 hat dieselbe Wirkung wie der längere Befehl MOVE.L #0,D0.

Nach dieser Initialisierung beginnt die eigentliche Arbeitsschleife beim Label "DoZiffer:". Das aktuelle Zeichen (beim ersten mal das erste Zeichen) wird aus dem Puffer ins Register D0 geholt und durch das Pluszeichen das Register um eins erhöht. Mit diesem Zeichen in D0 wird dann das Unterprogramm Value aufgerufen, das den passenden Wert zurückliefert.

Enthielt D0 kein gültiges Zeichen "a"-"z", "A"-"Z" oder "0"-"9", liefert die Unterroutine Value den Wert $FF zurück. Dies wird nach dem Aufruf sofort geprüft und gegebenenfalls zum Label "GetZahlFehler:" verzweigt. War aber alles korrekt, wird die Zahl zum Register D2 addiert. Die weitere Vorgehensweise hängt nun vom Zähler in D1 ab. Dieser wird nämlich um 1 vermindert und dann auf 0 geprüft. Enthält der Zähler den Wert 0, haben wir alle Ziffern behandelt und können die Umwandlung mit BEQ GetZahlOK beenden, andernfalls multiplizieren wir die bisherige Zahl in D2 mit 16 und beginnen erneut bei DoZiffer.

So wird die Zahl in D2 zusammengesetzt

Übrigens entspricht das Multiplizieren mit 16 genau einem Verschieben der bisherigen Zahl in D2 um eine Ziffer nach links. Dadurch wird also Platz für die nächste Ziffer geschaffen. Unser Programm macht also folgendes:

Wert in D0	Alter Wert in D2	Neuer Wert in D2
$0A	$00000000	$0000000A
$0B	$0000000A	$000000AB
$0C	$000000AB	$00000ABC

Zum Verschieben des Inhalts in D2 um eine Ziffer (4 Bit) benutzen wir den neuen Befehl ASL. Die Abkürzung bedeutet: Arithmetik Shift Left. Dieser Befehl verschiebt die Bits im Register um die angegebenen Stellenzahl nach links. Für die freiwerdenen Stellen rechts werden Null-Bits nachgeschoben.

Ist der Zähler in D1 heruntergezählt, wird der Inhalt von D2 nicht mehr verschoben, sondern das aktuelle Ergebnis durch einen Sprung zum Label "GetZahlOK:" in D2 und die Meldung "Alles klar ($FF)" in D0 zurückgeliefert.

Damit haben wir also ein Unterprogramm erstellt, mit dem wir über die Tastatur eingegebene Ziffernfolgen in Zahlen umwandeln können und können im nächsten Kapitel ein Programm entwickeln, mit dem wir den aktuellen Speicher des Amiga auf Diskette sichern können.

7.4 Speicherbereiche auf Diskette sichern

Das folgende Programm Savemem speichert einen beliebigen Teil des Amiga-Speichers auf Diskette. Dazu erfragt es eine Start- und eine Endadresse und sichert dann den angegebenen Bereich unter dem Dateinamen Savemem.dmp im aktuellen Verzeichnis.

Da Sie mit einer speziellen Option beim CLI-Befehl TYPE eine Datei als "Hexdump" anschauen können, können Sie sich mit diesem Programm über diesen Umweg den Speicher des Amiga anschauen. Doch kommen wir erst einmal zur Eingabe des Programms:

So erleichtern Sie sich die Arbeit beim Eingeben

Verwenden Sie als Basis für dieses Programm eins der schon vorher erstellten Programme, beispielsweise das Programm Codefile, und entfernen Sie alle nicht benötigten Elemente des Ausgangsprogramms. Fügen Sie anschließend an die entsprechende Stelle im Programm das im letzten Kapitel erstellte Unterprogramm GetZahl ein, in dem Sie den Cursor an die Stelle bewegen und den Befehl R verwenden. Entfernen Sie aus dem eingefügten Unterprogramm die beiden ILLEGAL-Befehle, und fügen Sie dort jeweils den Befehl RTS ein.

Das komplette Programm sieht dann folgendermaßen aus:

```
Start:

; **** Variablen definieren ****
Execbase = 4                ; Adresse der Exec.library
Mode_Newfile = 1006         ; Datei-Modus: neue Datei
OpenLibrary = -$0228        ; Offset
CloseLibrary = -$019E       ; Offset
Open = -$001E               ; Offset
Close = -$0024              ; Offset
Write = -$0030              ; Offset
Read = -$002A               ; Offset

; **** Hauptprogramm ****
JSR OpenDos                 ; Dos.library öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ Ende                    ; Nein, Ende
MOVE.L D0,Dosbase           ; sonst: Adresse merken
JSR OpenWin                 ; Fenster öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ CloseDos                ; Nein, Dos.library schließen
MOVE.L D0,Handle            ; sonst: Adresse merken

JSR GetStartadr             ;
CMP.L #0,D0                 ; Hat geklappt?
BEQ NoZahl                  ; Nein, Dos.library schließen
MOVE.L D2,Startbereich      ; merken
JSR GetEndadr               ;
CMP.L #0,D0                 ; Hat geklappt?
BEQ NoZahl                  ; Nein, Dos.library schließen
MOVE.L D2,Endbereich        ; merken
CMP.L Startbereich,D2       ;
BLT NoBereich               ; Start>Ende -> Fehler und schließen
JSR OpenFile                ; Ausgabe-Datei öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ NoDatei                 ; Fehler beim Dateiöffnen

JSR Speichern               ; Speicher in Datei schreiben
CMP.L #0,D0
BNE NoSchreiben             ; Nicht alles geschrieben->Fehler
JMP CloseAus                ; Alles schließen und Ende

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L #Dosname,A1          ; Libraryname nach 1
MOVE.L #0,D0                ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)         ; Interne Unterroutine
RTS

; **** Öffnen des Fensters ****
OpenWin:
```

```
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L #Fenster,D1         ; Fensterdefinition nach D1
MOVE.L #Mode_Newfile,D2    ; Access-Mode nach D2
JSR Open(A6)               ; Interne Unterroutine
RTS

; **** Text schreiben ****
Schreiben:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L Handle,D1           ; Fenster-Handle nach D1
SUB.L D2,D3                ; Bestimmung der Länge
JSR Write(A6)              ; Interne Unterroutine
RTS

; **** Zahl von Tastatur lesen ****
Lesen:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
MOVE.L Handle,D1           ; Fenster-Handle nach D1
MOVE.L #Puffer,D2          ; Platz für Name nach D2
MOVE.L #10,D3              ; Max 10 Zeichen lesen
JSR Read(A6)               ; Interne Unterroutine
RTS

GetStartAdr:
MOVE.L #TextZahl1,D2       ; Textanfang: "Startadresse"
MOVE.L #TextZahl1Ende,D3   ; Textende: "Startadresse"
JSR Schreiben              ; Text ausgeben
JSR Lesen                  ; Adresse eingeben lassen
CMP.L #9,D0                ; genau 8 Ziffern + Return ?
BNE GetStartAdrFehl        ; nein, falsche Eingabe
JSR GetZahl                ; Ziffern in Zahl umwandeln -> D2
RTS

GetStartAdrFehl:           ; Fehlerhafte Eingabe
MOVE.L #0,D0               ; Fehlermeldung nach D0
RTS                        ; Zurück zum Hauptprogramm

GetEndAdr:                 ; funktioniert wie "GetStartAdr"
MOVE.L #TextZahl2,D2
MOVE.L #TextZahl2Ende,D3
JSR Schreiben
JSR Lesen
CMP.L #9,D0
BNE GetEndAdrFehl
JSR GetZahl
RTS

GetEndAdrFehl:
MOVE.L #0,D0
RTS

; **** Öffnen der Ausgabedatei SaveMem.DMP ****
OpenFile:
MOVE.L Dosbase,A6          ; Dos-Basisadresse nach A6
```

```
MOVE.L #Dateiname,D1          ; Name nach D1
MOVE.L #Mode_Newfile,D2       ; Access-Mode "Neu" nach D2
JSR Open(A6)                  ; Interne Unterroutine
MOVE.L D0,HandleAus           ; Eingabehandle merken
RTS

GetZahl:
LEA Puffer,A0                 ; Pufferadresse holen
CLR.L D0
CLR.L D2                      ; Zahl = 0
MOVE.L #8,D1                  ; Zähler für Anzahl Ziffern
DoZiffer:
MOVE.B (A0)+,D0               ; Ziffer holen
JSR Value                     ; in Wert umwandeln
CMP.B #$FF,D0                 ; Fehler ?
BEQ GetZahlFehler             ; War keine Ziffer -> Fehler und Ende
ADD.L D0,D2                   ; Ziffer zu Zahl addieren
SUB.L #1,D1                   ; Ziffern fertig, Ende
BEQ GetZahlOK                 ; Ziffern fertig, Ende
ASL.l #4,D2                   ; Zahl * 16
JMP DoZiffer                  ; und weiter

GetZahlFehler:
MOVE.L #0,D0                  ; Fehlermeldung nach D0
RTS

GetZahlOK:
MOVE.L #$FF,D0                ; "Alles Klar" nach D0
RTS

Value: ; holt für eine Ziffer in D0 den Wert nach D0
CMP.B #'a',D0
BLT Value_Gross
CMP.B #'z',D0
BGT Value_Gross
CMP.B #'f',D0
BGT NoHex
SUB.B #'a',D0
ADD.B #10,D0
RTS

Value_Gross:
CMP.B #'A',D0
BLT Value_Ziffer
CMP.B #'Z',D0
BGT Value_Ziffer
CMP.B #'F',D0
BGT NoHex
SUB.B #'A',D0
ADD.B #10,D0
RTS

Value_Ziffer:
CMP.B #'0',D0
```

```
BLT NoHex
CMP.B #'9',D0
BGT NoHex
SUB.B #'0',D0
RTS

NoHex: MOVE.B #$FF,D0       ; Fehler, keine Hexziffer
RTS

; **** Abspeichern des Bereichs in die Datei ****
Speichern:
MOVE.L Dosbase,A6           ; Dos-Basisadresse nach A6
MOVE.L HandleAus,D1         ; Fenster-Handle nach D1
MOVE.L Startbereich,D2
MOVE.L Endbereich,D3
SUB.L Startbereich,D3       ; Ende-Start=Länge
JSR Write(A6)               ; Interne Unterroutine
MOVE.L Endbereich,D1
SUB.L Startbereich,D1       ; Ende-Start=Länge
SUB.L D1,D0                 ; Alles geschrieben ?
RTS


; **** Schließen der Ausgabedatei ****
CloseAus:
MOVE.L HandleAus,D1         ; Fenster-Handle nach D1
MOVE.L Dosbase,A6           ; Dos-Basisadresse nach A6
JSR Close(A6)               ; Interne Unterroutine

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Handle,D1            ; Fenster-Handle nach D1
MOVE.L Dosbase,A6           ; Dos-Basisadresse nach A6
JSR Close(A6)               ; Interne Unterroutine

; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L Dosbase,A1           ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)        ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Keine gültige Zahl, Fehler und Schließen ****
NoZahl:
MOVE.L #FehlerZahl,D2       ; Fehler: ungültige Zahl
MOVE.L #FehlerZahlEnde,D3
JSR Schreiben
JMP CloseWin                ; Fenster zu und Ende

; **** Kein gültiger Bereich, Fehler und Schließen ****
NoBereich:
```

```
MOVE.L #FehlerBereich,D2   ; Fehler:Bereich
MOVE.L #FehlerBereichEnde,D3
JSR Schreiben
JMP CloseWin               ; Fenster zu und Ende

; **** Fehler beim Öffnen der Datei ****
NoDatei:
MOVE.L #FehlerDatei,D2     ; Fehler:Bereich
MOVE.L #FehlerDateiEnde,D3
JSR Schreiben
JMP CloseWin               ; Fenster zu und Ende


; **** Fehler beim Schreiben der Datei ****
NoSchreiben:
MOVE.L #FehlerSchreiben,D2; Fehler:Bereich
MOVE.L #FehlerSchreibenEnde,D3
JSR Schreiben
JMP CloseAus               ; Alles schließen und Ende


; **** Speicherreservierung ****
EVEN
Dosbase:
DC.L 0                     ; Platz für Dos-Basisadresse

Handle:
DC.L 0                     ; Platz für Fenster-Handle

HandleAus:
DC.L 0                     ; Handle für Ausgabedatei

Startbereich: DC.L 0
Endbereich:   DC.L 0

Dosname:
DC.B `dos.library`,0       ; Name der Dos.library

Fenster:
DC.B `CON:10/10/600/180/FENSTER`,0 ; Fensterdefinition

TextZahl1:
DC.B `Startadresse (hex): `  ; Ausgabetext
TextZahl1ende:

TextZahl2:
DC.B `Endadresse   (hex): `  ; Ausgabetext
TextZahl2ende:

FehlerZahl:
DC.B `Fehler: 8 Stellen in Hex erwartet`,10  ; Ausgabetext
Fehlerzahlende:

FehlerBereich:
```

```
DC.B `Fehler: Start ist größer als Ende`,10  ; Ausgabetext
FehlerBereichende:

FehlerDatei:
DC.B `Fehler beim Dateiöffnen`,10  ; Ausgabetext
FehlerDateiende:

FehlerSchreiben:
DC.B `Datei-Schreib-Fehler!`,10    ; Ausgabetext
FehlerSchreibenEnde:

Dateiname: DC.B 'Savemem.DMP',0
DateinameEnde:
Puffer: DC.B '123456789',0,0
```

Speichern Sie das Programm nach der Eingabe unter dem Namen Savemem ab. Nach dem Starten wird ein Fenster geöffnet, in dem Sie nacheinander zur Eingabe einer Start- und Endadresse aufgefordert werden. Im Anschluß daran speichert das Programm den angegebenen Speicherbereich unter dem Dateinamen Savemem.DMP im aktuellen Verzeichnis ab.

So nutzen Sie das Programm vom CLI aus

Wenn Sie das Programm jederzeit vom CLI aus nutzen wollen, ändern Sie den Namen für das zu öffnende Fenster von "CON:" in "*" ab und ersetzen das ILLEGAL hinter dem Label "Ende:" durch ein RTS. Anschließend assemblieren Sie es erneut und speichern das lauffähige Programm mit dem Befehl WO unter dem Dateinamen Savemem ab.

Das Programm ist auch deswegen etwas umfangreicher geworden, weil es bei einigen möglichen Fehlern mit entsprechenden Fehlermeldungen abbricht. Folgende Fehler können gemeldet werden:

Fehler: 8 Stellen in Hex erwartet

Dieser Fehler tritt auf, wenn nicht acht Ziffern eingegeben wurden oder ein eingegebenes Zeichen keine gültige Hexziffer war.

Fehler: Start ist größer als Ende

Sie haben eine Endadresse kleiner als die Startadresse eingegeben.

Fehler beim Dateiöffnen

Die Datei Savemem.DMP zum Abspeichern des Amigaspeichers konnte nicht geöffnet werden.

Datei-Schreib-Fehler

Während des Schreibens in die Datei trat ein Fehler auf. Dieser Fehler tritt beispielsweise auf, wenn die Diskette oder RAM-Disk voll ist.

So erhält man das Betriebsystem auf Diskette

Das Betriebssystem des Amiga (ROM) befindet sich von Adresse $00FC0000 bis $00FFFFFF. Wenn Sie sich einen Teil des Betriebssytems in Ruhe anschauen wollen, können Sie folgendermaßen vorgehen:

1. Speichern Sie den gewünschten Speicherbereich mit Savemem in die RAM-Disk, beispielsweise von $00FC0000 bis $00FD0000. Dazu muß Savemem aus der RAM-Disk heraus gestartet werden.

2. Schauen Sie sich den Inhalt anschließend mit folgender Befehlszeile an:

```
TYPE RAM:SAVEMEM.DMP OPT H
```

Die Anzeige beginnt dann etwa folgendermaßen:

```
0000: 11114EF9 00FC00D2 0000FFFF 002100B4    ..N..........!..
0010: 002100C0 FFFFFFFF 65786563 2033332E    .!......exec 33.
0020: 31393220 2838204F 63742031 39383629    192 (8 Oct 1986)
0030: 0D0A0000 FFFFFFFF 0D0A0A41 4D494741    ...........AMIGA
0040: 20524F4D 204F7065 72617469 6E672053     ROM Operating S
0050: 79737465 6D20616E 64204C69 62726172    ystem and Librar
0060: 6965730D 0A436F70 79726967 68742028    ies..Copyright (
0070: 43292031 3938352C 20436F6D 6D6F646F    C) 1985, Commodo
```

```
0080: 72652D41 6D696761 2C20496E 632E0D0A    re-Amiga, Inc...
0090: 416C6C20 52696768 74732052 65736572    All Rights Reser
00A0: 7665642E 0D0A0000 65786563 2E6C6962    ved.....exec.lib
00B0: 72617279 00004AFC 00FC00B6 00FC323A    rary..J.......2:
```

Dies ist gerade der Beginn des Amiga-Betriebssystems mit Versionsnummer und Datum.

So funktioniert das Programm

Nach dem Öffnen der Dos.library und des Fensters, wird das Unterprogramm GetStartadr aufgerufen. Diese Unterroutine setzt D2 auf den Anfang des Textes "Startadresse (hex): " und D3 auf die Endadresse. Dann wird diese Eingabeaufforderung mit JSR Schreiben ausgegeben. Im Anschluß wird die Unterroutine Lesen dazu verwendet, die Ziffernfolge in den reservierten Puffer zu lesen. Im Anschluß daran prüft das Unterprogramm GetStart Adr, ob auch wirklich genau neun Zeichen eingegeben wurden, und springt, wenn dies nicht der Fall war, zur Fehlerroutine Get StartAdrFehl. Wurden aber neun Zeichen eingegeben, wird die Ziffernfolge mit der schon im letzten Kapitel entwickelten Unterroutine GetZahl in eine Zahl in D2 umgewandelt und zurückgeliefert. Das Unterprogramm GetStartAdr liefert also in D2 die Startadresse des zu speichernden Bereichs und in D0 einen Hinweis darauf, ob alles geklappt hat.

So stellen wir eine korrekte Eingabe sicher

Sie fragen sich jetzt vielleicht, warum wir nach dem Lesen von der Tastatur auf neun Zeichen prüfen (und nicht auf acht) und warum um alles in der Welt unser Unterprogramm Lesen sogar zehn Zeichen zuläßt (MOVE.L #10,D3)? Der Grund ist einfach: Es müssen neun Zeichen eingegeben werden, weil auch die Betätigung von <Return> nach der Eingabe als Zeichen zählt.

Das Unterprogramm Lesen muß zehn Zeichen zulassen, um feststellen zu können, ob zu viele Zeichen eingegeben wurden. Würden nämlich nur neun Zeichen zum Lesen zugelassen, könnte der Benutzer des Programms 30 Zeichen eingeben, nur würde das Betriebssystem nur die ersten neun Zeichen in den Puffer holen und auch nur neun Zeichen als eingegeben melden.

Wenn wir dann erneut Lesen aufrufen, würde uns das Betriebssytem weitere neun der schon vorher eingegebenen Zeichen in den Puffer schreiben, obwohl der Benutzer gar nichts eingegeben hat.

Diesen möglichen Fehler bei der Eingabe verhindern wir, in dem wir sofort zehn Zeichen zur Eingabe zulassen und prüfen, ob mehr als neun Zeichen (acht Ziffern und <Return>) eingegeben wurden.

So speichern wir den Bereich ab

Das Hauptprogramm prüft, ob bei der Eingabe und Umwandlung der Zahl ein Fehler aufgetreten ist und beendet gegebenenfalls das Programm mit einer entsprechenden Fehlermeldung. Ist aber alles glattgegangen, wird die ermittelte Zahl bei "Startbereich" abgelegt und anschließend das Unterprogramm Get Endadr aufgerufen. Dieses funktioniert identisch zum eben beschriebenen Unterprogramm, nur wird eine andere Eingabeaufforderung ausgegeben.

Nachdem beide Zahlen ermittelt und abgelegt wurden, prüft das Hauptprogramm, ob die Angaben überhaupt einen gültigen Bereich darstellen (Start < Ende). Ist dies nicht der Fall, wird die Fehlerroutine NoBereich aufgerufen, die nach einer entsprechenden Fehlermeldung endet.

Im Anschluß an die Bereichsprüfung versucht das Hauptprogramm, die Datei Savemem.DMP zu öffnen (JSR OpenFile). Auch hier wird eine Fehlerroutine aufgerufen, wenn das Öffnen der Datei nicht möglich war.

Zuletzt speichert unser Programm den Speicherbereich mit JSR Speichern in die Datei ab. Dabei wird einfach die Startadresse und als Länge Startadresse-Endadresse verwendet. Anschließend liefert das Betriebsystem ja in D0 zurück, wieviele Zeichen geschrieben wurden. Das Unterprogramm Speichern prüft nach, ob diese Anzahl genau der gewünschten Anzahl entspricht und liefert dementsprechen ein "OK" oder ein "Fehler" zurück. Auch hiernach prüft das Hauptprogramm, ob ein Fehler aufgetreten ist

und verzweigt gegebenenfalls zu einer entsprechenden Fehlerausgabe. Ging alles klar, ist die Aufgabe erledigt, und das Hauptprogramm verzweigt zur Routine CloseAus, die nacheinander alle geöffneten Dateien und Libraries schließt.

Grundregeln für längere Programme

Dieses Programm war zugegebenermaßen etwas länger als die bisherigen und daher auch mit mehr Zeitaufwand bei der Eingabe verbunden, aber dies war aus zwei Gründen notwendig:

Zum einen sollte das Programm wirklich nutzbar sein, zum anderen sollte es zeigen, wie man in einem umfangreicheren Programm Fehler erkennt und auf diese reagiert. Fassen wir noch einmal kurz zusammen:

- Das Programm besteht aus einem Hauptprogramm, die eigentliche "Arbeit" findet aber in Unterprogrammen statt. Ein Element des Hauptprogramms sieht also immer folgendermaßen aus:

```
JSR Unterprogramm        ; Wieder ein Stückchen Arbeit
CMP.L #0,D0              ; Ist ein Fehler aufgetreten?
BEQ Fehlerroutine        ; Ja, zur zugehörigen Fehlerroutine
                         ; ansonsten geht es hier weiter...
```

  Manche dieser Unterroutinen rufen weitere Unterprogramme auf, auch hierbei werden Fehler zurückgemeldet.

- Der Abbruch des Programms auf Grund eines Fehlers ist nun nicht mehr so einfach wie in den ersten kurzen Programmen. Wir können nicht mehr einfach ein Unterprogramm zum Schließen der Dos.library aufrufen und dann enden. Vielmehr müssen unter Umständen einige Libraries und Dateien geschlossen werden - in Abhängigkeit davon, wo der Fehler aufgetreten ist. Das Grundprinzip sieht dabei folgendermaßen aus:

```
Hauptprogramm:
JSR OpenLib1             ; erste Library öffnen
TST.L D0                 ; Fehler
BEQ Schließen1           ; ja, nur das geöffnete schließen
```

```
JSR OpenLib2                ; zweite Library öffnen
TST.L D0                    ; Fehler
BEQ Schließen2              ; ja, nur das geöffnete schließen

JSR OpenDatei               ; jetzt noch Datei öffnen
TST.L D0                    ; Fehler
BEQ Schließen3              ; ja, alles schließen

JSR Arbeit                  ; hier findet dann die Arbeit statt

JMP Schließen3              ; ab hier ebenfalls alles schließen

; **** Ende des Hauptprogramms ****

Schließen3:                 ; Das zuletzt Geöffnete schließen

Schließen2:                 ; Das Nächste schließen

Schließen1:                 ; Zuletzt das zuerst Geöffnete

Ende:                       ; Hier ist das Ende,
                            ; wenn nichts offen war

ILLEGAL                     ; Hier endet das Programm
```

7.5 So löscht man Dateien

Zum Schluß dieses Kapitel wollen wir Ihnen noch ein Beispiel dafür liefern, daß man mit Amiga-DOS (beziehungsweise der Dos.library) nicht nur Dateien öffnen, lesen und schreiben kann, sondern noch einiges mehr. So enthält die Dos.library etwa Unterprogramme zur Arbeit mit Verzeichnissen, zum Laden von Programmen und zur Veränderung von Dateien. Als Beispiel haben wir ein Programm zum Löschen einer Datei geschrieben. Dieses Programm nutzt das Unterprogramm DeleteFile der Dos.library.

```
Start:

; **** Variablen definieren ****
Execbase = 4                ; Adresse der Exec.library
Mode_Newfile = 1006         ; Datei-Modus: neue Datei
OpenLibrary = -$0228        ; Offset
CloseLibrary = -$019E       ; Offset
Open = -$001E               ; Offset
Close = -$0024              ; Offset
```

```
DeleteFile = -$0048       ; Offset
Write = -$0030            ; Offset
Read = -$002A             ; Offset

; **** Hauptprogramm ****
JSR OpenDos               ; Dos.library öffnen
CMP.L #0,D0               ; Hat geklappt?
BEQ Ende                  ; Nein, Ende
MOVE.L D0,Dosbase         ; sonst: Handle merken
JSR OpenWin               ; Fenster öffnen
CMP.L #0,D0               ; Hat geklappt?
BEQ CloseDos              ; Nein, Dos.library schließen
MOVE.L D0,Handle          ; sonst: Adresse merken
JSR Prompt                ; Eingabeaufforderung ausgeben
JSR Lesen                 ; Dateinamen holen
JSR Loeschen              ; Gewünschte Datei löschen
CMP.L #0,D0               ; Fehler beim Löschen ?
BEQ Fehler                ; ja, Fehlermeldung und Ende
JMP OK                    ; Nein, Meldung und Ende

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L #Dosname,A1        ; Libraryname nach 1
MOVE.L #0,D0              ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)       ; Interne Unterroutine
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Dosbase,A6         ; Dos-Basisadresse nach A6
MOVE.L #Fenster,D1        ; Fensterdefinition nach D1
MOVE.L #Mode_Newfile,D2   ; Access-Mode nach D2
JSR Open(A6)              ; Interne Unterroutine
RTS

; **** Text schreiben ****
Schreiben:
MOVE.L Dosbase,A6         ; Dos-Basisadresse nach A6
MOVE.L Handle,D1          ; Fenster-Handle nach D1
SUB.L D2,D3               ; Bestimmung der Länge
JSR Write(A6)             ; Interne Unterroutine
RTS

; **** Dateiname von Tastatur lesen ****
Lesen:
MOVE.L Dosbase,A6         ; Dos-Basisadresse nach A6
MOVE.L Handle,D1          ; Fenster-Handle nach D1
MOVE.L #Dateiname,D2      ; Platz für Name nach D2
MOVE.L #99,D3             ; Max 99 Zeichen lesen
JSR Read(A6)              ; Interne Unterroutine
MOVE.B D0,Dateinamelaenge ; Länge des Namens merken
RTS
```

```
; **** Eingabeaufforderung schreiben ****
Prompt:
MOVE.L #Text,D2              ; Speicher nach D2
MOVE.L #Textende,D3          ; Speicherende nach D3
JSR Schreiben                ; Unterprogramm
RTS

; **** Resultat ausgeben****
OK:
MOVE.L #OKText,D2            ; Speicher nach D2
MOVE.L #OKTextende,D3        ; Speicherende nach D3
JSR Schreiben                ; Unterprogramm
JMP CloseWin

; **** Fehlermeldung ausgeben ****
Fehler:
MOVE.L #FehlerText,D2        ; Speicher nach D2
MOVE.L #FehlerTextende,D3    ; Speicherende nach D3
JSR Schreiben                ; Unterprogramm
JMP CloseWin

; **** Löschen der angegebenen Datei****
Loeschen:
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
MOVE.L #Dateiname,D1         ; Fensterdefinition nach D1
CLR.L D2                     ;
MOVE.B Dateinamelaenge,D2    ; Länge des Namens holen
ADD.L D1,D2                  ; Startadresse hinzuzählen
SUB.L #1,D2                  ; 1 Byte weniger->letztes Byte, CR
MOVE.L D2,A0                 ; ins Adreßregister
MOVE.B #0,(A0)               ; 0 hineinschreiben, mit 0 abschließen
JSR DeleteFile(A6)           ; Interne Unterroutine
RTS

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Handle,D1             ; Fenster-Handle nach D1
MOVE.L Dosbase,A6            ; Dos-Basisadresse nach A6
JSR Close(A6)                ; Interne Unterroutine

; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6           ; Basisadresse nach A6
MOVE.L Dosbase,A1            ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)         ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Speicherreservierung ****
EVEN
```

```
Dosbase:
DC.L 0                      ; Platz für Dos-Basisadresse

Handle:
DC.L 0                      ; Platz für Fenster-Handle

Dosname:
DC.B `dos.library`,0        ; Name der Dos.library

Fenster:
DC.B `CON:10/10/600/180/FENSTER`,0  ; Fensterdefinition

Text:
DC.B `Bitte Dateinamen zum Löschen eingeben: `  ; Ausgabetext

Textende:                   ; Ende des Speichers

OKText:
DC.B `Datei gelöscht`,10    ; Ausgabetext

OKTextende:                 ; Ende des Speichers

FehlerText:
DC.B `Datei konnte nicht gelöscht werden! `,10  ; Ausgabetext

FehlerTextende:             ; Ende des Speichers

Dateiname:
Blk.B 100,0

Dateinamelaenge:
DC.B 0

Puffer:
DC.B 0                      ; Platz für ein Byte zum Dekodieren
```

Speichern Sie das Programm nach der Eingabe und dem Assemblieren unter dem Namen DelFile ab. Nach dem Start werden Sie nach einem Dateinamen gefragt, und diese Datei wird gelöscht. Dabei gibt das Programm am Schluß aus, ob das Löschen erfolgreich war oder nicht.

Bis auf das Unterprogramm Loeschen sind Ihnen alle Teile des Programms sicherlich bekannt, und Sie können das Programm daher auch leicht aus bestehenden Teilen zusammensetzen.

Hinweis: Die Vorgehensweise, immer wieder auf bestehende Programmteile zurückzugreifen, sollten Sie sich möglichst auch zu eigen machen. Zum einen können Sie

sich dadurch sehr viel Arbeit sparen, zum anderen verwenden Sie ja immer fehlerfreie, also schon getestete Programmteile. Würden Sie aber auch die Standardelemente immer neu schreiben, würden sich mit Sicherheit auch immer wieder Fehler einschleichen - eine unnötige Fehlerquelle.

8. Intuition

Langsam nähern wir uns dem Ende dieses Buches, und eigentlich sind Sie inzwischen auch gar kein Einsteiger mehr. Sie haben nämlich nicht nur einige Befehle des 68000 kennengelernt, sondern inzwischen sicherlich auch ein wenig Übung im Erstellen von Programmen. Allerdings wollen wir dieses Buch keineswegs abschließen, ohne Sie mit einer äußerst interessanten Library (Bibliothek) bekannt zu machen - der Intuition.library.

Im Gegensatz zur bisherigen Programmierung müssen wir uns nämlich mit zwei Besonderheiten vertraut machen, und dabei wollen wir Ihnen natürlich im Rahmen dieses Einsteigerbuches helfen und von vornherein Fehler und Frust vermeiden. Bei der Nutzung der Intuition.library müssen wir uns an zweierlei gewöhnen:

1. Bei unseren bisherigen Beispielen haben wir praktisch die gesamte Kontrolle über unser Programm und seine Objekte (Variablen, Dateien usw.) gehabt. Wir haben ein (einfaches) Fenster geöffnet, einen Text ausgegeben und dann eine Eingabe verlangt. Bei der Arbeit mit Intuition erstellt man komplexere Objekte (z.B. Fenster) und übergibt einen Teil der Kontrolle an Intuition. Dann wartet man auf bestimmte Ereignisse und reagiert dementsprechend.

2. Natürlich müssen die komplexeren Objekte auch präzise beschrieben werden. Bei einer Datei war das noch ganz einfach, da brauchte man im wesentlichen nur den Dateinamen. Schon bei einem Fenster sieht das aber ganz anders aus. Hier erwartet Intuition 18 (!) Angaben von uns.

   Für das Übergeben aller dieser Angaben kann man natürlich nicht mehr die Prozessorregister verwenden. Statt dessen verwendet man Tabellen - in der Programmiersprache C als Strukturen bekannt. Leider ist die Arbeit mit solchen Strukturen in C auch etwas einfacher als im SEKA. Wenn

man aber einmal das Grundprinzip verstanden hat, kann man mit Intuition sehr interessante und ansprechende Programme erstellen.

Hinweis: Bisher konnte uns beim Programmieren nur dann etwas passieren, wenn wir einen Befehl falsch verwendet haben. Bei der Arbeit mit Intuition sieht das etwas anders aus. Ein Programm kann auch deshalb abstürzen, weil eine Tabelle, die ein Objekt beschreibt, fehlerhaft ist. Prüfen Sie also bei den folgenden Programmen die Daten und dort besonders die Anzahl und Länge (Byte, Wort oder Langwort) der einzelnen Angaben sehr genau. Ein versehentliches "DC.W" statt eines "DC.B" bringt Intuition meist völlig durcheinander und führt zum Absturz des Amiga.

8.1 Wir öffnen ein Intuition-Fenster

Im ersten Versuch mit Intuition wollen wir uns noch etwas beschränken und nicht alle notwendigen Vorgänge wirklich durchführen. Statt dessen wollen wir uns darauf beschränken, ein Intuition-Window zu erstellen und nach einigen Sekunden wieder zu schließen. Wie üblich erfolgt die Erklärung des Programms am Ende.

```
Start:

; **** Variablen definieren ****
Execbase = 4                ; Adresse der Exec.library
OpenLibrary = -$0228        ; Offset in Exec.library
CloseLibrary = -$019E       ; Offset in Exec.library
Open = -$001E               ; Offset in Dos.library
Close = -$0024              ; Offset in Dos.library
Delay = -$00C6              ; Offset in Dos.library
OpenWindow = -$00CC         ; Offset in Intuition.library
CloseWindow= -$0048         ; Offset in Intuition.library

; **** Hauptprogramm ****
JSR OpenDos                 ; Dos.library öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ Ende                    ; Nein, Ende
```

```
MOVE.L D0,Dosbase           ; sonst: Adresse merken
JSR OpenIntui               ; Intuition.library öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ CloseDos                ; Nein, Ende
MOVE.L D0,Intuibase         ; sonst: Adresse merken
JSR OpenWin                 ; Fenster öffnen
CMP.L #0,D0                 ; Hat geklappt?
BEQ CloseIntui              ; Nein, Ende
MOVE.L D0,Winhandle         ; sonst: Nummer merken
JSR Warten
JMP CloseWin                ; Fenster, Intui und Dos schließen, Ende

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L #Dosname,A1          ; Libraryname nach A1
MOVE.L #0,D0                ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen der Intuition.library ****
OpenIntui:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L #Intuiname,A1        ; Libraryname nach A1
MOVE.L #0,D0                ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Intuibase,A6         ; Intuition-Basisadresse nach A6
LEA NewWindow,A0            ; Fensterdefinition nach A0
JSR OpenWindow(A6)
RTS

; **** Warteschleife ****
Warten:
MOVE.L #200,D1              ; Wartezeit nach D1
MOVE.L Dosbase,A6           ; Basisadresse nach A6
JSR Delay(A6)               ; Warteroutine 1/50 Sek.
RTS

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Winhandle,A0         ; Fenster-Handle nach A0
MOVE.L Intuibase,A6         ; Intuition-Basisadresse nach A6
JSR CloseWindow(A6)         ; Interne Unterroutine

; **** Schließen der Intuition.library ****
CloseIntui:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L Intuibase,A1         ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)        ; Interne Unterroutine
```

```
; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L Dosbase,A1           ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)        ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Speicherreservierung ****
EVEN
NewWindow:
DC.W 10                     ; X
DC.W 10                     ; Y
DC.W 100                    ; Breite
DC.W 100                    ; Höhe
DC.B 1                      ; Schrift weiß
DC.B 3                      ; Hintergrund rot
DC.L 0                      ; IDCMP-Flags (Meldungen: keine)
DC.L $1000!1!2!4!8          ; Activate+Size+Drag+Depth+Close
DC.L 0                      ; Kein First Gadget
DC.L 0                      ; Keine Grafik für Checkmark
DC.L Windowtitel            ; Adresse des Titels für das Fenster
DC.L 0                      ; Adresse des Screens
DC.L 0                      ; Adresse der Bitmap für Window
DC.W 20                     ; Minimale Breite
DC.W 20                     ; Minimale Höhe
DC.W 629                    ; Maximale Breite
DC.W 246                    ; Maximale Höhe
DC.W 1                      ; Screentyp Workbench

Dosbase:
DC.L 0                      ; Platz für Dos-Basisadresse

Intuibase:
DC.L 0                      ; Platz für Intuition-Basisadresse

Winhandle:
DC.L 0                      ; Platz für Window-Kennung

Dosname:
DC.B `dos.library`,0        ; Name der Dos.library

Intuiname:
DC.B `intuition.library`,0; Name der Library

Windowtitel:
DC.B `Intuition-Fenster`,0
```

Starten des Programms

Nach der Eingabe und fehlerfreien Assemblierung speichern Sie das Programm bitte vor dem Starten mit dem Befehl W ab. Als Namen können Sie beispielsweise IWindow (für Intuition-Window) angeben. Nun kommt der entscheidende Augenblick: Starten Sie das Programm mit G ohne Angabe von Breakpoints. Wenn alles glatt geht, erscheint für vier Sekunden ein kleines Fenster in der oberen, linken Ecke des Bildschirms - wenn nicht der Guru. Prüfen Sie im zweiten Fall noch einmal genau die Tabelle mit der Fensterbeschreibung.

Öffnen der Intuition.library

Bevor wir mit Intuition arbeiten können, müssen wir natürlich die Intuition.library öffnen. Der Vorgang entspricht bis auf den anderen Namen der Bibliothek genau dem Öffnen der Dos.library. Das Hauptprogramm prüft wie üblich, ob die Bibliothek geöffnet werden konnte und verzweigt gegebenenfalls zum Programmende. Hat aber alles geklappt, wird die Basisadresse der Intuition.library in der Variablen Intuibase abgelegt.

Öffnen und Schließen des Fensters

Zum Öffnen und Schließen von Intuition-Fenstern dienen zwei spezielle Unterprogramme OpenWindow und CloseWindow, deren Offsets zu Beginn unseres Programms definiert wurden. (Verwechseln Sie diese nicht mit unseren Unterprogrammen OpenWin und CloseWin!) Der Unterroutine zum Öffnen des Fensters wird als einziger Parameter die Adresse der Fensterdefinition (Tabelle) übergeben. Das Unterprogramm liefert eine neue Adresse zurück, vergleichbar dem Dateihandle nach dem Öffnen einer Datei. Dieses muß später zum Schließen des Fensters der Routine CloseWindow übergeben werden.

Einen Unterschied zwischen einem Dateihandle und einem Fensterhandle gibt es aber doch: Das Dateihandle, das von der Dos.library nach Open zurückgeliefert wird, können wir ausschließlich an andere DOS-Unterprogramme (also Unterroutinen der Dos.library) übergeben. Das Fensterhandle ist aber gleichzei-

tig die Adresse einer neuen Tabelle, in der wir viele interessante Informationen über unser Fenster erfahren können. Schon im übernächsten Programm werden wir Ihnen zeigen, wie man eine Information aus dieser Tabelle holt und weiterverwendet.

Die Definition des Fensters

Kommen wir nun zum eigentlich aufwendigen Teil beim Öffnen eines Intuition-Windows: der Fensterdefinition. Sie beginnt am Label "NewWindow:":

```
DC.W 10              ; X
DC.W 10              ; Y
DC.W 100             ; Breite
DC.W 100             ; Höhe
```

Die ersten vier Angaben sind jeweils ein Wort (0-65535) lang und beschreiben die Position und Größe des Fensters. Dabei werden die Koordinaten der linken, oberen Ecke des Fensters und die Ausmaße angegeben. Wenn das Fensteröffnen einmal nicht klappt und die Unterroutine Openwindow eine Null zurückliefert, haben Sie wahrscheinlich Angaben gemacht, die nicht möglich sind, beispielsweise eine Breite von 1000 Punkten angegeben. Der Workbench-Screen hat eine Breite von 640 Punkten und ist 256 Punkte hoch. Folglich dürfen X + Breite nicht mehr als 640 und Y + Höhe nicht mehr als 256 ergeben. Um ein sehr großes Fenster zu erstellen, können Sie also beispielsweise für Breite 629 und als Höhe 246 angeben.

```
DC.B 1               ; Schrift weiß
DC.B 3               ; Hintergrund rot
```

Die nächsten beiden Angaben sind nur ein Byte lang und bestimmen die Farben beim Zeichnen oder Schreiben in das Fenster. Da der Workbench-Screen vier Farben hat, sind hier Angaben zwischen 0 und 3 möglich. Unsere Kommentare beziehen sich übrigens auf die Standardeinstellungen, die Sie in den Preferences einstellen können. Wenn Sie dort Änderungen vornehmen, erhält das Fenster auch andere Farben.

Übrigens bestimmt der Hintergrund nicht etwa die Grundfarbe des Fensters, sondern nur, welche Farbe beim Schreiben in das Fenster nicht gesetzte Punkte erhalten. Da der Fenstertitel schon mit diesen Farben geschrieben wird, erscheint dort die Schrift weiß auf rotem Untergrund.

```
DC.L 0                    ; IDCMP-Flags (Meldungen: keine)
```

Dieses Langwort gibt an, welche Ereignisse uns von Intuition gemeldet werden sollen. Wir hatten ja zu Beginn dieses Kapitels schon darauf hingewiesen, daß man Objekte gewissermaßen der Kontrolle durch Intuition übergibt. Natürlich möchte man dann über bestimmte Ereignisse informiert werden. Da unser Beispiel noch möglichst einfach sein soll, wollen wir noch nicht auf Nachrichten von Intuition reagieren müssen und geben hier eine Null an.

Hinweis: Für dieses und das nächste Langwort gilt: Jedes Bit im Langwort steht für eine ganz bestimmte Aufgabe. Soll diese Aufgabe ermöglicht werden, muß das Bit gesetzt werden. Mehrere Aufgaben können kombiniert werden, in dem die zugehörigen Bits gesetzt und der so entstehende Wert angegeben wird.

```
DC.L $1000!1!2!4!8        ; Activate+Size+Drag+Depth+Close
```

Dieses Langwort bestimmt, welche Möglichkeiten und Besonderheiten unser Fenster erstellt. Wir haben eine spezielle Schreibweise verwendet, die die genutzten Möglichkeiten deutlich zeigt: die Oder-Verknüpfung. Werden zwei Zahlen Oder-verknüpft, sind im Ergebnis alle die Bits gesetzt, die in einer der beiden oder in beiden Zahlen gesetzt waren. Das Zeichen für die Oder-Verknüpfung im SEKA ist das Ausrufezeichen. Beispiel:

```
4 ! 2 = %00000100 ! %00000010 = %00000110 = 6
```

In unserem Fall ergibt sich also als Ergebnis für das Langwort: $0000100F. Die einzelnen Komponenten haben dabei folgende Auswirkungen:

$1000 AKTIVATE: Gibt an, daß unser Fenster nach dem Öffnen automatisch aktiviert sein soll.

$0001 WINDOWSIZING: Nur wenn dieses Bit gesetzt ist, kann das Fenster in der Größe geändert werden.

$0002 WINDOWDRAG: Ermöglicht das Verschieben des Fensters auf dem Bildschirm.

$0004 WINDOWDEPTH: Erlaubt es, das Fenster nach vorn oder hinten zu klicken. Wird dieses Bit nicht gesetzt, fehlen die beiden zugehörigen Gadgets in der oberen, rechten Ecke des Fensters.

$0008 WINDOWCLOSE: Versieht das Fenster mit einem Schließsymbol. Dadurch kann es aber noch nicht geschlossen werden, Sie können also ruhig einmal auf dieses Symbol klicken. Wie man mit dem Schließsymbol ein Schließen des Fensters ermöglicht, zeigen wir im nächsten Kapitel.

```
DC.L 0                    ; Kein First Gadget
DC.L 0                    ; Keine Grafik für Checkmark
```

Diese beiden Langworte ermöglichen spezielle Anpassungen des Fensters und werden von uns nicht weiter behandelt.

```
DC.L Windowtitel          ; Adresse des Titels für das Fenster
```

In diesem Langwort muß die Adresse des Fenstertitels angegeben werden. Der Titel muß wie üblich mit Null abgeschlossen sein.

```
DC.L 0                    ; Adresse des Screens
DC.L 0                    ; Adresse der Bitmap für Window
```

Das erste Langwort ermöglicht es, das Fenster nicht auf dem Workbenchscreen, sondern auf einem selbsterzeugten Screen zu öffnen. Dann müßte hier die Adresse des Screens (also sozusagen das Screen-Handle) eingetragen werden. Wir bleiben aber auf dem Workbench-Screen. Das zweite Langwort dient für spezielle Anwendungen.

```
DC.W 20                     ; Minimale Breite
DC.W 20                     ; Minimale Höhe
DC.W 629                    ; Maximale Breite
DC.W 246                    ; Maximale Höhe
```

Mit diesen vier Angaben in Wort-Länge werden die möglichen Veränderungen in der Größe festgelegt. Intuition sorgt selbständig für die Einhaltung dieser Werte.

```
DC.W 1                      ; Screentyp Workbench
```

Hier muß der Typ des Screens angegeben werden, auf dem das Fenster erscheinen soll. Wir benutzen den Wert WORKBENCHSCREEN (= 1), hier könnte auch CUSTOMSCREEN (= 15) festgelegt werden.

Hinweis: Vielleicht wundern Sie sich darüber, daß wir die Begriffe für manche Werte in Großbuchstaben schreiben, beispielsweise WORKBENCHSCREEN. Dies geschieht immer, wenn wir uns auf festgelegte Definitionen beziehen. Diese Werte finden Sie beispielsweise in den speziellen Include-Dateien (Erweiterung ".h", also beispielsweise Screens.h) eines C-Compilers oder (Erweiterung ".i", also beispielsweise Screens.i) eines Assemblers oder im "Amiga Intern" von DATA BECKER.

8.2 Wir überprüfen das Schließsymbol

Kommen wir nun zu einer Erweiterung des Programms, die auch gleich den richtigen Umgang mit einem Intuition-Fenster zeigt. Wir werden das Fenster nun nicht mehr automatisch nach einer festgelegten Zeit schließen, sondern erst, wenn das Schließsymbol betätigt wurde.

Bevor wir das Programm abdrucken und erläutern, wollen wir kurz die richtigen Schritte und die Reihenfolge bei der Arbeit mit einem Objekt in Intuition am Beispiel eines Fensters vorstellen:

1. Zuerst muß das in der Tabelle definierte Fenster mit Open-Window geöffnet werden. Zurück erhält man von Intuition ein Window-Handle, das auch gleichzeitig die Adresse einer neuen Window-Tabelle ist.

2. Durch das Öffnen des Fensters hat man gleichzeitig die Überwachung des Fensters an Intuition übertragen. Allerdings wird man über alle Ereignisse informiert, die man in der Fensterdefinition angegeben hat. Man holt sich aus der neuen Tabelle (ab Window-Handle) die Adresse eines speziellen Messageports für dieses Fenster. Ein Messageport ist eine Art Briefkasten, in dem Nachrichten abgelegt werden. Mit einem speziellen Unterprogramm aus der Exec.library WaitPort wartet man darauf, daß eine Nachricht vorhanden ist.

   Bei diesem "Warten" verschwendet man übrigens keine Rechenzeit, im Gegensatz zu einer Schleife, die ständig nachfragt, ob schon etwas angekommen ist. Dies ist ähnlich wie bei der Funktion Delay aus der Dos.library, die im Gegensatz zu einer Schleife in unserem Programm ebenfalls keine unnötige Rechenzeit kostet.

3. Ist eine Nachricht angekommen, reagiert man entsprechend darauf. Hat man mehrere Ereignisse angegeben, über die man informiert werden möchte, muß man sich die Nachricht natürlich genauer anschauen. Wir werden aber in unserem folgenden Beispiel nur auf eine einzige Art von Nachricht warten: der Betätigung des Schließsymbols.

4. Zum Schluß wird das Intuition-Fenster natürlich wieder ordnungsgemäß geschlossen.

```
Start:

; **** Variablen definieren ****
Execbase = 4                ; Adresse der Exec.library
OpenLibrary = -$0228        ; Offset
CloseLibrary = -$019E       ; Offset
Open = -$001E               ; Offset
Close = -$0024              ; Offset
Delay = -$00C6              ; Offset zum Warten
```

```
OpenWindow = -$00CC       ; Offset
CloseWindow= -$0048       ; Offset
WaitPort = -$0180

; **** Hauptprogramm ****
JSR OpenDos               ; Dos.library öffnen
CMP.L #0,D0               ; Hat geklappt?
BEQ Ende                  ; Nein, Ende
MOVE.L D0,Dosbase         ; sonst: Adresse merken
JSR OpenIntui             ; Intuition.library öffnen
CMP.L #0,D0               ; Hat geklappt?
BEQ CloseDos              ; Nein, Ende
MOVE.L D0,Intuibase       ; sonst: Adresse merken
JSR OpenWin               ; Fenster öffnen
CMP.L #0,D0               ; Hat geklappt?
BEQ CloseIntui            ; Nein, Ende
MOVE.L D0,Winhandle       ; sonst: Nummer merken
JSR Warten
JMP CloseWin              ; Fenster, Intui und Dos schließen, Ende

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L #Dosname,A1        ; Libraryname nach A1
MOVE.L #0,D0              ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen der Intuition.library ****
OpenIntui:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L #Intuiname,A1      ; Libraryname nach A1
MOVE.L #0,D0              ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Intuibase,A6       ; Intuition-Basisadresse nach A6
LEA NewWindow,A0          ; Fensterdefinition nach A0
JSR OpenWindow(A6)
RTS

; **** Warteschleife ****
Warten:
MOVE.L Winhandle,A0       ; Fensterkennung
MOVE.L 86(A0),A0          ; Message-Port für Benutzer
MOVE.L ExecBase,A6        ;
JSR WaitPort(A6)          ; Nachricht vorhanden ist
MOVE.L D0,A1
MOVE.L 20(A1),D1          ; Ereignis nach D1 holen
RTS

; **** Schließen des Fensters ****
```

```
CloseWin:
MOVE.L Winhandle,A0         ; Fenster-Handle nach A0
MOVE.L Intuibase,A6         ; Intuition-Basisadresse nach A6
JSR CloseWindow(A6)         ; Interne Unterroutine

; **** Schließen der Intuition.library ****
CloseIntui:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L Intuibase,A1         ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)        ; Interne Unterroutine


; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L Dosbase,A1           ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)        ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Speicherreservierung ****

EVEN
NewWindow:
DC.W 10                     ; X
DC.W 10                     ; Y
DC.W 300                    ; Breite
DC.W 100                    ; Höhe
DC.B 3                      ; Schrift rot (4. Farbe)
DC.B 2                      ; Hintergrund schwarz (3. Farbe)
DC.L $200                   ; IDCMP-Flags (Meldung: CloseWindow)
DC.L $1000!1!2!4!8          ; Activate+Size+Drag+Depth+Close
DC.L 0                      ; Kein First Gadget
DC.L 0                      ; Keine Grafik für Checkmark
DC.L Windowtitel            ; Adresse des Titels für das Fenster
DC.L 0                      ; Adresse des Screens
DC.L 0                      ; Adresse der Bitmap für Window
DC.W 20                     ; Minimale Breite
DC.W 20                     ; Minimale Höhe
DC.W 640                    ; Maximale Breite
DC.W 250                    ; Maximale Höhe
DC.W 1                      ; Screentyp Workbench

Dosbase:
DC.L 0                      ; Platz für Dos-Basisadresse

Intuibase:
DC.L 0                      ; Platz für Intuition-Basisadresse

Winhandle:
DC.L 0                      ; Platz für Window-Kennung
```

```
Dosname:
DC.B `dos.library`,0       ; Name der Dos.library

Intuiname:
DC.B `intuition.library`,0; Name der Library

Windowtitel:
DC.B `Intuition-Fenster`,0
```

Speichern Sie das Programm nach dem Assemblieren am besten unter dem Namen Winklick ab. Im wesentlichen hat sich neben der Definition eines neuen Unterprogramms WaitPort und einer Änderung in der Fenstertabelle nur das Unterprogramm Warten geändert.

Die Funktionsweise des Programms

Ein wichtiger Unterschied in der Fenstertabelle besteht darin, daß wir nun ein Ereignis angegeben haben, bei dem wir informiert werden wollen: $200 = WINDOWCLOSE, also Betätigung des Schließsymbols. Hätten wir hier weiterhin eine Null für "keine Benachrichtigung" angegeben, könnten wir in unserem Unterprogramm Warten "bis ans Ende unserer Tage" warten.

Im Unterprogramm benutzen wir nun einen speziellen Mechanismus, um ohne Verschwendung von Prozessorzeit auf die Betätigung des Schließsymbol zu warten: Wir warten bis zu dem Zeitpunkt, an dem in "unserem Briefkasten" eine Nachricht vorhanden ist. Dazu benötigen wir natürlich die Adresse dieses Briefkastens, des sogenannten Message-Ports. Diese Adresse befindet sich in der Fenstertabelle ab Adresse 86 als Langwort.

Wir holen also zuerst die Adresse der Fenstertabelle (das Windowhandle) nach A0 und dann den Inhalt des Langwortes ab Offset 86 ebenfalls nach A0, wo ihn das anschließend aufgerufene Unterprogramm WaitPort erwartet. Aus diesem Unterprogramm kehrt der Prozessor erst wieder in unser Programm zurück, wenn das Schließsymbol betätigt wurde und Intuition uns folglich eine Nachricht im Briefkasten hinterlegt hat.

Diese Nachricht (eigentlich eher die Adresse einer Nachrichtentabelle) liefert uns die Routine wie üblich in D0 zurück. Ab

Offset 20 finden wir dann ein Langwort, das genau so aufgebaut ist, wie unser Langwort in der Fenstertabelle, mit dem die gewünschten Nachrichten festgesetzt wurden. In unserem Fall enthält das Langwort also den Wert $200, und wir sichern den Wert in D1, auch wenn wir ihn hier nicht direkt weiterverwenden. Anschließend endet unser Unterprogramm Warten.

Haben Sie Spaß an Nachrichten und Reaktionen gefunden? Dann probieren Sie doch einmal folgende Programmänderung aus: Ersetzen Sie in der Fenstertabelle die Zeile:

```
DC.L $200                ; IDCMP-Flags (Meldung: CloseWindow)
```

durch folgende neue Zeile:

```
DC.L $200 ! $10000       ; IDCMP (Meldung: CloseWindow, ???)
```

Assemblieren Sie das Programm erneut, und starten Sie es. Betätigen Sie diesmal nicht das Schließsymbol (dies ist natürlich weiterhin möglich), sondern entfernen Sie eine Diskette aus dem Laufwerk.

Wir haben dadurch als zusätzliche Meldung $10000 = DISKREMOVED angegeben und erhalten nun jeweils eine Meldung, wenn die Diskette aus dem Laufwerk entfernt wurde - keine schlechte Idee beispielsweise für einen selbstprogrammierten Filerequester.

Hinweis: Jetzt ist es natürlich auch wichtig, zu prüfen, was für eine Nachricht denn vorhanden ist, nun können wir ja zwei verschiedenen Nachrichten erhalten. Wir könnten unser Unterprogramm Warte also beispielsweise folgendermaßen erweitern:

```
; **** Warteschleife ****
Warten:
MOVE.L Winhandle,A0 ; Fensterkennung
MOVE.L 86(A0),A0    ; Message-Port für Benutzer
MOVE.L ExecBase,A6  ;
JSR WaitPort(A6)    ; Warten, bis Nachricht vorhanden ist
MOVE.L D0,A1
MOVE.L 20(A1),D1    ; Ereignis nach D1 holen
```

```
CMP.L #$200,D1          ; Schließsymbol?
BEQ WartenEnde          ; Ja, fertig
JSR LED_Blink           ; LED-Blinken lassen
JMP Warten              ; und weiter warten

WartenEnde:
RTS
```

Sie müssen dann natürlich noch das Unterprogramm zum Blinken der LED aus einem unserer ersten Programme einfügen. Dann wird bei jedem Entfernen einer Diskette die LED blinken, und beim Betätigen des Schließsymbols das Programm enden. Übrigens können Sie auch das Einlegen einer Diskette abfragen, der Wert für DISKINSERTED ist $8000.

8.3 Wie schreibt man Texte in ein Fenster?

Haben Sie sich schon einmal überlegt, ob Sie wohl mit der Funktion WRITE der Dos.library Texte im Intuition-Fenster ausgeben können, wie in einem einfachen DOS-Fenster? Die Anwort lautet: leider nein. Sie sehen, bei DOS-Fenstern ist vieles einfacher, aber auch längst nicht so interessant und leistungsfähig, denn die Mindest- und Maximalgrößen für ein DOS-Fenster können wir nicht festlegen, und ein Schließsymbol gibt es auch nicht.

Trotzdem wollen Sie sicherlich wissen, wie man denn in einem Intuition-Fenster einen Text ausgeben kann. Die Antwort ist einfach und für die Arbeit mit Intuition geradezu typisch: Man erstellt eine Tabelle, die wichtige Informationen über den Text und die Art der Ausgabe enthält und ruft die Intuition-Routine PrintIText auf. Das folgende Programm zeigt die richtige Vorgehensweise, am Schluß werden dann wieder alle Neuerungen beschrieben.

```
Start:

; **** Variablen definieren ****
Execbase = 4                 ; Adresse der Exec.library
OpenLibrary = -$0228         ; Offset in Exec.library
```

```
CloseLibrary = -$019E      ; Offset in Exec.library
Open = -$001E              ; Offset in Dos.library
Close = -$0024             ; Offset in Dos.library
Delay = -$00C6             ; Offset in Dos.library
OpenWindow = -$00CC        ; Offset in Intuition.library
CloseWindow= -$0048        ; Offset in Intuition.library
WaitPort = -$0180          ; Offset in Exec.library
PrintIText = -$00D8        ; Offset in Intuition.library

; **** Hauptprogramm ****
JSR OpenDos                ; Dos.library öffnen
CMP.L #0,D0                ; Hat geklappt?
BEQ Ende                   ; Nein, Ende
MOVE.L D0,Dosbase          ; sonst: Adresse merken
JSR OpenIntui              ; Intuition.library öffnen
CMP.L #0,D0                ; Hat geklappt?
BEQ CloseDos               ; Nein, Ende
MOVE.L D0,Intuibase        ; sonst: Adresse merken
JSR OpenWin                ; Fenster öffnen
CMP.L #0,D0                ; Hat geklappt?
BEQ CloseIntui             ; Nein, Ende
MOVE.L D0,Winhandle        ; sonst: Nummer merken
JSR PRINT                  ; Text ausgeben
JSR Warten                 ; und warten
JMP CloseWin               ; Fenster, Intui und Dos schließen, Ende

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L #Dosname,A1         ; Libraryname nach A1
MOVE.L #0,D0               ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen der Intuition.library ****
OpenIntui:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L #Intuiname,A1       ; Libraryname nach A1
MOVE.L #0,D0               ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Intuibase,A6        ; Intuition-Basisadresse nach A6
LEA NewWindow,A0           ; Fensterdefinition nach A0
JSR OpenWindow(A6)
RTS

; **** Warteschleife ****
Warten:
MOVE.L Winhandle,A0        ; Fensterkennung
MOVE.L 86(A0),A0           ; Message-Port für Benutzer
MOVE.L ExecBase,A6         ;
```

```
JSR WaitPort(A6)          ; Warten, bis Nachricht vorhanden ist
MOVE.L D0,A1
MOVE.L 20(A1),D1          ; Ereignis nach D1 holen
RTS

; **** Text im Fenster ausgeben ****
PRINT:
MOVE.L Intuibase,A6       ;
MOVE.L Winhandle,A0       ; Windowtabelle
MOVE.L 50(A0),A0          ; Rastport in Windowtabelle
LEA Windowtext,A1         ; Texttabelle
MOVE.L #20,D0             ; X-Offset
MOVE.L #20,D1             ; Y-Offset
JSR PrintIText(A6)        ; Text ausgeben
RTS

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Winhandle,A0       ; Fenster-Handle nach A0
MOVE.L Intuibase,A6       ; Intuition-Basisadresse nach A6
JSR CloseWindow(A6)       ; Interne Unterroutine

; **** Schließen der Intuition.library ****
CloseIntui:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L Intuibase,A1       ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)      ; Interne Unterroutine


; **** Schließen der Dos.library ****
CloseDos:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L Dosbase,A1         ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)      ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Speicherreservierung ****

EVEN
WindowText:
DC.B 1,2                  ; Farben: Text, Hintergrund
DC.B 0                    ; Zeichenmodus normal
EVEN
DC.W 20                   ; Position X
DC.W 20                   ; Position Y
DC.L 0                    ; Standard-Font
DC.L Textpuffer           ; Adresse des Textes
DC.L 0                    ; Weitere Texttabelle: keine

NewWindow:
DC.W 10                   ; X
```

```
DC.W 10                     ; Y
DC.W 300                    ; Breite
DC.W 100                    ; Höhe
DC.B 1                      ; Schrift weiß
DC.B 3                      ; Hintergrund rot
DC.L $200                   ; IDCMP-Flags (Meldung: CloseWindow)
DC.L $1000!1!2!4!8          ; Activate+Size+Drag+Depth+Close
DC.L 0                      ; Kein First Gadget
DC.L 0                      ; Keine Grafik für Checkmark
DC.L Windowtitel            ; Adresse des Titels für das Fenster
DC.L 0                      ; Adresse des Screens
DC.L 0                      ; Adresse der Bitmap für Window
DC.W 20                     ; Minimale Breite
DC.W 20                     ; Minimale Höhe
DC.W 640                    ; Maximale Breite
DC.W 250                    ; Maximale Höhe
DC.W 1                      ; Screentyp Workbench

Dosbase:
DC.L 0                      ; Platz für Dos-Basisadresse

Intuibase:
DC.L 0                      ; Platz für Intuition-Basisadresse

Winhandle:
DC.L 0                      ; Platz für Window-Kennung

Dosname:
DC.B `dos.library`,0        ; Name der Dos.library

Intuiname:
DC.B `intuition.library`,0; Name der Library

Windowtitel:
DC.B `Intuition-Fenster`,0

Textpuffer: DC.B `Ein Text im Intuition-Fenster`,0
```

Speichern Sie das Programm beispielsweise unter Wintext ab. Nach dem Start erscheint unser bekanntes Fenster und etwa in der Mitte des Fensters unser Text. Nachdem Sie den Erfolg ausreichend betrachtet haben, können Sie das Programm mit einem Klick auf das Schließsymbol beenden.

Der Aufbau der Tabelle für den Text

Unsere Tabelle zur Definition des auszugebenen Textes beginnt beim Label "WindowText:". Zuerst werden drei Bytes angegeben: Die ersten beiden bestimmen die Zeichenfarbe für Vorder- und

Hintergrund - wir kennen sie schon aus der Fenstertabelle. Das nächste Byte gibt einen Zeichenmodus an. Hier kann zusätzlich festgelegt werden, ob beispielsweise alle nicht gesetzten Punkte der Schrift mit der Hintergrundfarbe beschrieben werden sollen, oder ob der Text invertiert ausgegeben werden soll. Durch die Angabe der Null legen wir fest, daß nur die gesetzten Punkte der Schrift am Bildschirm erscheinen, nicht gesetzte Punkte dagegen nicht beachtet werden.

Als nächstes muß die X- und Y-Position angegeben werden. Vergessen Sie vorher keinesfalls den Befehl EVEN, denn die Position wird als Wort angegeben und muß an einer geraden Adresse liegen. Übrigens bestimmen X und Y nicht vollständig die Position des auszugebenden Textes, weil die Routine PrintIText zusätzlich in zwei Registern noch einen Offset erwartet. Dieser wird zu den hier angegebenen Werten hinzugezählt.

Zum Schluß muß die Adresse des Textes angegeben werden, und optional ist dahinter die Angabe einer weiteren Adresse mit einer Textstruktur möglich. Durch die Null geben wir an, daß keine mehr folgt.

So wird der Text ausgegeben

Die Unterroutine PrintIText erwartet vier Angaben beim Aufruf. Zuerst ist der sogenannte Rastport gefragt. Ein Rastport ist - wie könnte es beim Amiga anders sein - erst einmal wieder eine Tabelle. Er enthält wichtige Informationen über die Zeichenfläche, so beispielsweise die Adressen der zuhörigen Speicherbereiche (der Bitmaps) und damit auch die Größen, die Zeichenstifte und vieles mehr. Wir brauchen an dieser Stelle nur zu wissen, daß die Adresse des Rastports in der Fenstertabelle ab Offset 50 zu finden ist. Zusätzlich benötigt das Unterprogramm natürlich noch die Adresse der Texttabelle und - wie schon bei der Erläuterung der Tabelle angesprochen - einen X- und Y-Offset. Diese werden zu den Positionsangaben in der Tabelle hinzugezählt.

Nach all den Vorbereitungen können wir endlich das Unterprogramm PrintIText aufrufen und zum Hauptprogramm zurückkehren. Dort wird dann die bekannte Unterroutine Warten aufgerufen.

8.4 Wir malen Punkte und Striche

Haben Sie noch Spaß an Fenstern und ähnlichem? Dann wollen wir Ihnen zumindest beim Einstieg in eine Fülle neuer Unterprogramme behilflich sein. Hat man nämlich erst einmal ein Fenster und damit eine Zeichenfläche, den sogenannten Rastport, stehen eine Vielzahl nützlicher Unterprogramme aus der Graphics.library offen. Zwei Beispiele wollen wir in unserem nächsten Programm vorstellen.

Hinweis: Vielleicht waren Sie schon einmal etwas enttäuscht, weil wir für einen Anwendungsbereich, der Sie besonders interessiert hat, nur eins von vielen möglichen Beispielen gebracht haben. Dies reicht aber völlig aus, denn haben Sie erst einmal die grundsätzliche Vorgehensweise verstanden, brauchen Sie nur noch bestimmte Informationen, um auch alle anderen Möglichkeiten nutzen zu können. Und wie Sie an diese Informationen gelangen und sie ausnutzen können, erfahren Sie im Kapitel "Aufsteigen zum Amiga-Profi".

```
Start:

; **** Variablen definieren ****
Execbase = 4                ; Adresse der Exec.library
Rastport = 50               ; Offset Rastport in Fenstertabelle
OpenLibrary = -$0228        ; Offset in Exec.library
CloseLibrary = -$019E       ; Offset in Exec.library
Open = -$001E               ; Offset in Dos.library
Close = -$0024              ; Offset in Dos.library
Delay = -$00C6              ; Offset in Dos.library
OpenWindow = -$00CC         ; Offset in Intuition.library
CloseWindow= -$0048         ; Offset in Intuition.library
WaitPort = -$0180           ; Offset in Exec.library
WritePixel = -$0144         ; Offset in Graphics.library
Move = -$00F0               ; Offset in Graphics.library
```

```
Draw = -$00F6              ; Offset in Graphics.library

; **** Hauptprogramm ****
JSR OpenDos                ; Dos.library öffnen
CMP.L #0,D0                ; Hat geklappt?
BEQ Ende                   ; Nein, Ende
MOVE.L D0,Dosbase          ; sonst: Adresse merken
JSR OpenIntui              ; Intuition.library öffnen
CMP.L #0,D0                ; Hat geklappt?
BEQ CloseDos               ; Nein, Ende
MOVE.L D0,Intuibase        ; sonst: Adresse merken
JSR OpenGfx                ; Graphics.library öffnen
CMP.L #0,D0                ; Hat geklappt?
BEQ CloseIntui             ; Nein, Ende
MOVE.L D0,Gfxbase          ; sonst: Adresse merken
JSR OpenWin                ; Fenster öffnen
CMP.L #0,D0                ; Hat geklappt?
BEQ CloseGfx               ; Nein, Ende
MOVE.L D0,Winhandle        ; sonst: Nummer merken
JSR Malen                  ;
JSR Warten
JMP CloseWin               ; Fenster, Intui und Dos schließen, Ende

; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L #Dosname,A1         ; Libraryname nach A1
MOVE.L #0,D0               ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen der Intuition.library ****
OpenIntui:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L #Intuiname,A1       ; Libraryname nach A1
MOVE.L #0,D0               ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen der Graphics.library ****
OpenGfx:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L #Grafikname,A1      ; Libraryname nach A1
MOVE.L #0,D0               ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS

; **** Öffnen des Fensters ****
OpenWin:
MOVE.L Intuibase,A6        ; Intuition-Basisadresse nach A6
LEA NewWindow,A0           ; Fensterdefinition nach A0
JSR OpenWindow(A6)
RTS
```

```
; **** Punkt und Linie im Fenster zeichnen ****
Malen:
MOVE.L GfxBase,A6          ;
MOVE.L Winhandle,A0        ; Windowadresse holen
MOVE.L Rastport(A0),A1     ; Rastport in Windowtabelle
MOVE.L #50,D1              ; Y-Position
MOVE.L #50,D0              ; X-Position
MOVE.L Winhandle,A0        ; Windowadresse holen
MOVE.L Rastport(A0),A1     ; Rastport in Windowtabelle
JSR WritePixel(A6)         ; Punkt zeichnen
MOVE.L Winhandle,A0        ; Windowadresse holen
MOVE.L Rastport(A0),A1     ; Rastport in Windowtabelle
MOVE.L #10,D1              ; Y-Position
MOVE.L #10,D0              ; X-Position
JSR Move(A6)               ; Zeichenstift auf Anfang setzen
MOVE.L Winhandle,A0        ; Windowadresse holen
MOVE.L Rastport(A0),A1     ; Rastport in Windowtabelle
MOVE.L #200,D1             ; Y-Position
MOVE.L #80,D0              ; X-Position
JSR Draw(A6)               ; Linie zeichnen

RTS
; **** Warteschleife ****
Warten:
MOVE.L Winhandle,A0        ; Fensterkennung
MOVE.L 86(A0),A0           ; Message-Port für Benutzer
MOVE.L ExecBase,A6         ;
JSR WaitPort(A6)           ; Warten, bis Nachricht vorhanden ist
MOVE.L D0,A1
MOVE.L 20(A1),D1           ; Ereignis nach D1 holen
RTS

; **** Schließen des Fensters ****
CloseWin:
MOVE.L Winhandle,A0        ; Fenster-Handle nach A0
MOVE.L Intuibase,A6        ; Intuition-Basisadresse nach A6
JSR CloseWindow(A6)        ; Interne Unterroutine

; **** Schließen der Intuition.library ****
CloseGfx:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L Gfxbase,A1          ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)       ; Interne Unterroutine

; **** Schließen der Intuition.library ****
CloseIntui:
MOVE.L Execbase,A6         ; Basisadresse nach A6
MOVE.L Intuibase,A1        ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)       ; Interne Unterroutine


; **** Schließen der Dos.library ****
CloseDos:
```

```
MOVE.L Execbase,A6          ; Basisadresse nach A6
MOVE.L Dosbase,A1           ; Dos-Basisadresse nach A1
JSR CloseLibrary(A6)        ; Interne Unterroutine

; **** Beendigen des Programms ****
Ende:
ILLEGAL

; **** Speicherreservierung ****

EVEN
Randomwert: DC.L $12345678

NewWindow:
DC.W 10                     ; X
DC.W 10                     ; Y
DC.W 300                    ; Breite
DC.W 100                    ; Höhe
DC.B 1                      ; Schrift weiß
DC.B 3                      ; Hintergrund rot
DC.L $200                   ; IDCMP-Flags (Meldung: CloseWindow)
DC.L $1000!1!2!4!8          ; Activate+Size+Drag+Depth+Close
DC.L 0                      ; Kein First Gadget
DC.L 0                      ; Keine Grafik für Checkmark
DC.L Windowtitel            ; Adresse des Titels für das Fenster
DC.L 0                      ; Adresse des Screens
DC.L 0                      ; Adresse der Bitmap für Window
DC.W 20                     ; Minimale Breite
DC.W 20                     ; Minimale Höhe
DC.W 255                    ; Maximale Breite
DC.W 255                    ; Maximale Höhe
DC.W 1                      ; Screentyp Workbench

Dosbase:
DC.L 0                      ; Platz für Dos-Basisadresse

Intuibase:
DC.L 0                      ; Platz für Intuition-Basisadresse

Gfxbase:
DC.L 0                      ; Platz für Grafik-Basisadresse

Winhandle:
DC.L 0                      ; Platz für Window-Kennung

Dosname:
DC.B `dos.library`,0        ; Name der Dos.library

Intuiname:
DC.B `intuition.library`,0; Name der Library

Grafikname:
DC.B `graphics.library`,0 ; Name der Library
```

```
Windowtitel:
DC.B `Intuition-Fenster`,0
```

Speichern Sie das Programm nach dem fehlerfreien Assemblieren beispielsweise unter dem Namen Winpunkt ab. Das Programm zeichnet einen Punkt und eine Linie in das geöffnete Fenster und wartet anschließend, bis Sie das Schließsymbol betätigen.

So funktioniert das Programm

Zunächst wird eine weitere Bibliothek im Amiga geöffnet, die Graphics.library. Diese stellt uns eine Fülle graphischer Möglichkeiten zur Verfügung, aus denen wir hier nur das Setzen eines Punktes und Zeichnen einer Linie herausgegriffen haben. Auf die notwendigen Programmzeilen zum Öffnen der Graphics.library und zur Ablage der Basisadresse brauchen wir wohl kaum noch einzugehen, das kennen Sie nun schon aus vielen vorherigen Programmen. Interessanter ist da schon das Unterprogramm Malen:

So zeichnet man einen Punkt

Ist die Graphics.library einmal geöffnet, ist das Punktezeichnen beinahe ein Kinderspiel. Man holt aus der Fenstertabelle ab Offset 50 den Rastport nach A1, schreibt die X-Position in D0 und die Y-Position und ruft das Unterprogramm WritePixel auf. (Ein Pixel ist übrigens ein Graphikpunkt.) Bleibt nur ein kleiner Hinweis: Gezeichnet wird mit den aktuellen Farben für Vorder- und Hintergrund - für unser Beispiel kein Problem. Möchten Sie dagegen diese Farben für Punkte unterschiedlich setzen, dann können Sie folgendes Unterprogramm der Graphics.library verwenden:

```
SetAPen = -$0156
```

Das Unterprogramm erwartet in A1 die Adresse des Rastports und in D0 die Nummer der gewünschten Farbe.

So zeichnet man eine Linie

Um eine Linie zu zeichnen, bewegen wir den Zeichenstift mit der Unterroutine Move an die in D0 und D1 angegebene Startposition. In A1 muß die Adresse des Rastports vorhanden sein. Dies ist zwar in unserem Fall noch vom Zeichnen des Punktes der Fall, aber sicher ist sicher - wir holen uns besser die Adresse erneut aus der Fenstertabelle.

Anschließend schreiben wir die Position der Endpunkte für die Linie in D0 und D1 und rufen mit dem Rastport in A1 das Unterprogramm Draw auf, dieses zieht eine Linie von der aktuellen Position des Zeichenstiftes bis zur angegebenen Position.

9. Aufsteigen zum Amiga-Profi

Haben Sie inzwischen Spaß an Maschinensprache gefunden? Wir hoffen es zumindest. In diesem Kapitel wollen wir Ihnen zweierlei zeigen:

1. Wie kommt man an die notwendigen Informationen heran? Wie erfährt man beispielsweise die Adresse der LED, oder den Offset der EXEC-Unterroutine OpenLibrary, oder die Bits, mit denen ein Intuition-Fenster gestaltet werden kann?

2. Wie kann man Programme und Informationen für die Programmiersprache C für eigene Assemblerprogramme verwenden.

Ganz konkret zeigen wir die Vorgehensweise dann an zwei Beispielprogrammen, bei denen wir genau zeigen, wie wir auf die Ideen und Informationen gekommen sind. Zum Schluß dieses Kapitels haben Sie dann die Voraussetzungen, um den Amiga ganz zu beherrschen und Dinge möglich zu machen, von denen andere nur träumen.

9.1 Wie nutzt man Informationen aus einem Intern

Bisher haben Sie alle wichtigen Informationen von uns bekommen. Jetzt wollen wir Ihnen verraten, aus welchen Quellen wir unsere Informationen schöpfen und was man dazu wissen sollte.

Die folgenden Informationen beziehen sich auf das Amiga Intern von DATA BECKER. Sollten Sie eine andere Auflage besitzen, können natürlich sowohl Seiten- und Kapitelangaben als auch Inhalte geändert sein. Schauen Sie im Inhaltsverzeichnis oder im Index nach, um die Informationen in Ihrer Ausgabe zu finden.

Zum jetzigen Zeitpunkt plant DATA BECKER ein neues Amiga Intern, in dem die bisherigen Bände zu einem Band zusammengefaßt werden. Sollten Sie diesen Band besitzen, fällt die Unterteilung in die beiden Einzelbände weg.

Der Aufbau des Amiga Intern

Beginnen wir mit dem grundsätzlichen Aufbau des Amiga Intern. Es besteht aus drei Teilen: Der erste Teil beschreibt die Hardware des Amiga, Teil 2 die Exec.library und deren Aufgaben und der dritte Teil Amiga-DOS und die Dos.library.

Im folgenden Abschnitt wollen wir Ihnen einige Beispiele dafür bringen, wie man Informationen aus dem Intern für eigene Programme verwenden kann.

Beispiele für die Hardware

Erinnern Sie sich noch an unsere ersten Erfahrungen mit der LED des Amiga? Im Amiga-Intern-Kapitel "CIA 8520" unter der Zwischenüberschrift "Die Einbindung der CIAs in das Amiga-System" finden wir eine Tabelle "CIA-A: Registeradressen". Dort werden auch die Bits der Adresse $BFE001 beschrieben, unter D1 (für Bit 1) finden Sie den Hinweis auf die LED.

Auch ein zweites Beispiel aus unserem Einsteigerbuch läßt sich als Information im Intern wiederfinden: das Programm zum Feststellen von Mausbewegungen. Im Amiga Intern-Kapitel "Maus, Joystick und Paddles" ist das Register für die Maus in Game-Port 0 angegeben: JOY0DAT = Adresse $DFF00A. Während wir uns im Programm ausschließlich auf Änderungen in dieser Speicherstelle bezogen haben, finden Sie dort auch genaue Informationen darüber, wie eine Änderung der Werte zu interpretieren ist.

Beispiele für Exec

Eine Fülle von Informationen finden Sie in der sogenannten "Exec-Base-Struktur". Beispielsweise befindet sich in der Tabelle

ab Offset 420 ($1A4) eine Liste aller wartenden Tasks im Amiga, ab Offset 406 ($196) eine Liste aller Tasks, die jetzt Prozessorzeit möchten. Sie wissen sicherlich, daß im Amiga mehrere Programme praktisch gleichzeitig ablaufen und Tasks genannt werden. Wenn Sie nicht genau wissen, wie eine Liste aufgebaut ist, finden Sie die Informationen unter dem Stichwort "Listen". Gut für uns ist auch, daß jeweils gleich die Offsets angegeben sind, so daß wir mit den Informationen sofort ein Programm schreiben können. Um beispielsweise an den Namen des ersten wartenden Task heranzukommen, sind folgende Befehlszeilen notwendig:

```
MOVE.L 4,A6             ; Execbase nach A6
MOVE.L 420(A6),A0       ; Anfang der Liste der wartenden Task
                        ; = Node des ersten Task
MOVE.L 10(A0),D0        ; D0 enthält Adresse des Namens
```

Interessant ist im Amiga Intern auch das Kapitel "Multitasking". Dort erfahren wir beispielsweise, wie die einzelnen Tasks aufgebaut sind und verwaltet werden. Zusätzlich finden wir alle Funktionen der Exec.library zur Verwaltung von Tasks, so daß sich damit interessante Möglichkeiten für eigene Programme ergeben. Ein Beispiel für die Nutzung dieser Informationen zeigt das Programm STOP in Kapitel 9.3 des vorliegenden Buches.

Beispiele für Amiga-DOS

Eine Erläuterung aller in der Dos.library vorhandenen Funktionen finden Sie im Amiga Intern im Kapitel "Die DOS-Bibliothek". Einige der Unterprogramme haben Sie ja schon in unseren Beispielprogrammen kennengelernt.

Bisher haben unsere Programme die Parameter nachträglich über ein eigenes CON-Fenster oder das CLI-Fenster erfragt. Die CLI-Befehle statt dessen erhalten ihre Parameter direkt beim Aufruf. Wie man an diese Parameter aus der Kommandozeile herankommt, zeigt das Kapitel "Programme" im Amiga Intern. Wir werden diese Informationen im Programm COMP in Kapitel 10.4 verwenden.

Mit Amiga-DOS können nicht nur Dateien geöffnet, gelesen oder geschrieben werden. Vielmehr ist auch der direkte Zugriff auf Disketten möglich. Wenn Sie also in einem eigenen Programm Sektoren der Diskette lesen oder schreiben wollen, finden Sie die zugehörigen Informationen in Kapitel "Trackdisk-Device: Zugriff auf Disketten". Dort steht beispielsweise auch ein Maschinenspracheprogramm zum Lesen von Sektoren, daß Sie schnell in eigene Programme einbauen können.

So verwenden Sie Amiga Intern Band II

Brauchen Sie genauere Informationen zu allen (!) Bibliotheken im Amiga und den darin enthaltenen Unterroutinen? Dann liefert Ihnen Intern Band II im Kapitel "Die Amiga Libraries" die notwendigen Informationen.

Erinnern Sie sich noch an unser Intuition-Fenster? In der Tabelle für das zu erstellende Fenster konnte in einem Langwort angegeben werden, welche Komponenten und Möglichkeiten das Fenster haben sollte. Unsere Definitionszeile sah folgendermaßen aus:

```
DC.L $1000!1!2!4!8       ; Activate+Size+Drag+Depth+Close
```

Welche weiteren Möglichkeiten Sie noch angeben können, erfahren Sie beispielsweise im Intern Band II unter "Window_Flags" bei der Beschreibung der Funktion OpenWindow im Kapitel "Die Intuition-Library".

Erinnern Sie sich daran, daß wir zur Überprüfung des Schließsymbols an den Message-Port (Briefkasten) unseres Fensters gelangen mußten. Seine Adresse steht in der Fensterstruktur, die Intuition beim Öffnen des Fensters erzeugt und deren Anfangsadresse uns von der Unterroutine OpenWindow zurückgeliefert wird, ab Offset 86. Und der für das Malen von Punkten und Linien wichtige Rastport (unsere Zeichenfläche) befindet sich an der Position 50. In der Tabelle gibt es aber noch viel mehr Informationen, beispielsweise die aktuelle Größe des Fensters oder die Position der Maus. Alle Inhalte der Fenstertabelle werden beispielsweise im Intern unter "struct Window" erläutert.

9.2 Wie setzt man Informationen für C um

Nachdem wir Ihnen nun an einigen Beispielen erläutert haben, wie man an die interessanten und notwendigen Informationen gelangt, wollen wir Ihnen natürlich auch bei der optimalen Nutzung dieser Informationen helfen. Viele Angaben beziehen sich nämlich nicht auf Maschinensprache, sondern beispielsweise auf C. Daher sollte man einige Besonderheiten der Sprache C zumindest kennen, damit man diese Informationen auch für Maschinensprache verwenden kann.

Grundsätzlich gibt es drei Probleme bei der Umsetzung von C-Programmen und -Informationen für Maschinensprache:

Die Include-Dateien

Die meisten C-Programme (wie sie beispielsweise in Zeitschriften abgedruckt werden), beginnen etwa mit folgenden Zeilen:

```
#INCLUDE <EXEC/TYPES.H>
#INCLUDE <EXEC/EXECBASE.H>
#INCLUDE <EXEC/TASKS.H>
```

Dadurch erhält das Programm zusätzliche Informationen aus speziellen Dateien, die beim Compilieren des Programms eingefügt (englisch: to include) werden. Leider sind diese Dateien beim Abdruck des Programms nicht sichtbar, so daß wir die Angaben für Assembler nicht so einfach anschauen können.

Wenn Sie einen C-Compiler besitzen, ist die Lösung für Sie kein Problem. Dort werden ja die Include-Dateien mitgeliefert, und Sie können sich die notwendigen Informationen leicht anschauen. Beim Aztec 3.6 befinden sie sich beispielsweise auf der ersten Diskette im Verzeichnis INCLUDE.

Haben Sie allerdings keinen C-Compiler, bleiben Ihnen zumindest noch die Informationen aus dem Amiga Intern Band II. Dort werden bei den jeweiligen Bibliotheksfunktionen auch die zugehörigen Include-Informationen abgedruckt. Beispielsweise beim Öffnen eines Fensters mit OpenWindow ist sowohl die

notwendige Struktur NewWindow abgedruckt, deren Adresse der Funktion OpenWindow übergeben werden muß, als auch die Definition der einzelnen Bits für die Definiton des Fensters (Window_Flags).

Was sind eigentlich Strukturen

Da wir gerade beim Stichwort Strukturen sind: Diese sind in C sehr beliebt und bei der Programmierung des Amigas auch sehr häufig nötig. Deshalb sollten wir zumindest wissen, was Strukturen sind und wie man mit ihnen umgeht.

Eine Struktur ist grundsätzlich eine Tabelle. Unsere Tabelle zur Definition eines Intuition-Fensters ist also eine Struktur. Allerdings sind viele der für den Amiga wichtigen Strukturen schon in den vorher erläuterten Include-Dateien vordefiniert. Diese Definition beschreibt ganz präzise den Aufbau der jeweiligen Tabelle. So enthält die Tabelle für das Intuition-Fenster beispielsweise Bytes, Worte und Langworte, und das wird durch die Definition in C festgelegt.

Schauen wir uns also den Anfang der Struktur in C (entnommen aus Amiga Intern) und unsere Tabelle aus dem Beispielprogramm Winklick an:

```
struct NewWindow
{
SHORT LeftEdge;
SHORT TopEdge;
SHORT Width;
SHORT Height;
UBYTE DetailPen
UBYTE BlockPen
ULONG IDCMPFlags

NewWindow:
DC.W 10                   ; X
DC.W 10                   ; Y
DC.W 300                  ; Breite
DC.W 100                  ; Höhe
DC.B 3                    ; Schrift rot (4. Farbe)
DC.B 2                    ; Hintergrund schwarz (3. Farbe)
DC.L $200                 ; IDCMP-Flags (Meldung: CloseWindow)
```

Durch die Angabe "struct NewWindow" wird eine Struktur mit dem angegebenen Namen eingeleitet, die geschweifte Klammer gibt an, daß jetzt die einzelnen Definitionen, also die Tabelleneinträge, folgen.

Der jeweils erste Begriff gibt die Größe des Eintrags an. So bedeutet SHORT beispielsweise ein Wort (also zwei Bytes), und deshalb haben wir an die entsprechende Stelle den Assemblerbefehl DC.W geschrieben. Eine Liste der C-Begriffe zur Definition der Variablenlängen finden Sie weiter unten.

Als nächstes folgt in der C-Definition der Name des Tabelleneintrags. Um etwas Vergleichbares auch in Assembler zu erhalten, hätten wir beispielsweise ein Label vor den Tabelleneintrag setzen können:

```
NewWindow:
LeftEdge: DC.W 10          ; X
```

Allerdings gibt es zwischen der C-Definition und unserer Tabelle einen großen Unterschied: Die C-Definition beschreibt nur das Aussehen der Tabelle, legt aber diese noch nicht im Speicher des Amiga an wie unsere Tabelle.

Um die Tabelle in C auch im Speicher anzulegen, ist folgende Zeile notwendig:

```
struct NewWindow NeuesFenster =
{
10,
10,
300,
100,
3,
... (und so weiter)
```

Dadurch wird die Tabelle also mit den angegebenen Werten im Speicher abgelegt.

Hinweis: Bei einigen Assemblern (nicht beim SEKA) und auch beim Aztec-C-Compiler werden spezielle Include-Dateien für Assembler mitgeliefert. Diese haben statt

der Dateierweiterung ".h" für C-Programme die Erweiterung ".i". Schauen Sie im Handbuch oder auf den Disketten nach, ob das bei Ihnen auch der Fall ist, denn diese Dateien mit ihren Definitionen können die Anpassung von Programmen sehr erleichtern und viel Schreibarbeit sparen helfen.

Wie werden Bibliotheksroutinen in C aufgerufen?

Anschließend wird das Fenster dann mit folgender Befehlszeile geöffnet:

```
Winhandle = OpenWindow(&NeuesFenster)
```

Dies entspricht in etwa dem Aufruf unseres Unterprogramms "JSR OpenWin". Die in den Klammern angegebenen Variablen werden von C automatisch in die richtigen Register geschrieben, in diesem Fall also in das Register A0. Das Zeichen "&" in C entspricht in etwa unserem "#". Es soll also nicht der Inhalt der Speicherstelle "NeuesFenster", sondern deren Adresse verwendet werden, wie wir es ja auch mit der Befehlszeile "LEA NewWindow,A0" machen. Wir hätten dafür auch schreiben können "MOVE.L #NewWindow,A0".

Hinweis: Einige Assembler können die Zeile "MOVE.L #NewWindow,A0" (also das Laden einer Adresse in ein Adressregister) so nicht verarbeiten und benötigen statt dessen die Befehlszeile mit LEA.

Der Rückgabewert der Funktion OpenWindow wird in C durch das Gleichheitszeichen in der Variablen "Winhandle" abgelegt. Dasselbe macht ja unser Hauptprogramm mit der Befehlszeile "MOVE.L D0,Winhandle".

Nun müssen wir noch verstehen, wie in C auf die einzelnen Elemente einer Struktur zugegriffen wird, denn immerhin haben Sie ja durch die Definition einen Namen bekommen. Eine Befehlszeile in C könnte etwa folgendermaßen aussehen:

```
NeuesFenster.LeftEdge = 20
```

Das bedeutet in etwa: Setze in der Tabelle NeuesFenster den Tabelleneintrag LeftEdge auf den neuen Inhalt 20. Da wir in unserem Beispiel in der Tabelle diesem Eintrag das Label "LeftEdge:" gegeben haben, können wir dies ebenfalls leicht erreichen durch:

```
MOVE.W 20,LeftEdge
```

Dabei wissen wir, daß wir als Länge ".W" nehmen müssen, weil der Tabelleneintrag LeftEdge ja als SHORT definiert worden ist.

Raucht Ihnen schon der Kopf? Wir könnten es zumindest verstehen, denn so langsam schreiben wir fast ein "C für Einsteiger" und kein "Maschinensprache für Einsteiger". Aber mit ein bißchen Grundwissen über C-Programme können wir viele wichtige Informationen für unsere eigenen Programme erhalten. Kommen wir daher zu den beiden letzten für uns wichtigen Besonderheiten.

Erinnern Sie sich noch an unser Problem, daß wir von Intuition beim Öffnen des Fensters eine neue Adresse erhalten, die Adresse der von Window verwalteten Fenstertabelle und aus dieser den Messageport ermitteln mußten? Unsere Zeilen in Maschinensprache sahen dafür folgendermaßen aus:

```
MOVE.L Winhandle,A0      ; Fensterkennung
MOVE.L 86(A0),A0         ; Message-Port für Benutzer
MOVE.L ExecBase,A6       ;
JSR WaitPort(A6)         ; Warten, bis Nachricht vorhanden ist
```

Die vergleichbaren Zeilen in C könnten etwa so aussehen:

```
Briefkasten = Winhandle->UserPort
WaitPort(Briefkasten)
```

Was ist nun der Unterschied zwischen folgenden beiden Befehlszeilen in C:

```
NeuesFenster.LeftEdge = 20

Briefkasten = Winhandle->UserPort
```

Im ersten Fall haben wir eine Tabelle im Speicher erzeugt. Auf die einzelnen Elemente wird dann mit dem Namen der erzeugten Struktur, einem Punkt und dem Namen des Tabelleneintrags zugegriffen.

Haben wir aber nur die Adresse einer Struktur erhalten (in unserem Beispiel die Adresse der Fenstertabelle), dann benutzen wir statt des Punktes den Pfeil "->". In beiden Fällen ist aber wichtig, daß wir den Offset des entsprechenden Tabelleneintrags kennen. Bei allen Strukturen, die im Amiga Intern Band II abgedruckt sind, ist dieser Offset glücklicherweise dezimal und hexadezimal angegeben.

Die Größen von Variablen

Ein Problem bei der Umsetzung von C-Programmen und deren Informationen in Assembler ist die Größe der Variablen. In Maschinensprache wird diese jeweils direkt beim Befehl angegeben (beispielsweise ".b" oder ".l"). In C werden diese statt dessen bei der Festlegung der Variable bestimmt. Wird diese Variable später im Programm verwendet, benutzt der Compiler automatisch die richtige Befehlslänge. Häufig kommen die folgenden Variablentypen vor:

char	Byte	.b
int	Wort, 2 Bytes	.w
short	Wort, 2 Bytes	.w
long	Langwort, 4 Bytes	.l

Was sind eigentlich Zeiger?

Zusätzlich spielen in C noch Zeiger eine große Rolle. Diese werden bei der Definiton durch ein Sternchen "*" vor dem Variablennamen festgelegt, beispielsweise durch:

```
int *Breite
```

Dadurch wird eine Adresse festgelegt, an der sich eine Variable vom Typ int (also ein Wort oder zwei Bytes) befindet. Adressen

sind im Amiga immer vom Typ long, also ein Langwort lang. Ein Zugriff auf diese Adresse in C sieht dann beispielsweise folgendermaßen aus:

```
*Breite = 100
```

Das bedeutet: Hole die Adresse von "Breite" (also ein Langwort), und schreibe dort das Wort 100 hin, also in Assembler:

```
LEA Breite,A0
MOVE.W #100,(A0)
```

Zusammenfassung

Zugegebenermaßen waren einige dieser Informationen für jemanden, der überhaupt noch keinen Kontakt mit der Sprache C hatte, nicht ganz einfach zu verstehen. Trotzdem waren Sie unserer Meinung nach sehr wichtig, weil viele Informationen über den Amiga für die Sprache C geschrieben sind und man ungefähr wissen sollte, wie diese in Assembler umgesetzt werden. Halten wir noch einmal die wichtigsten Informationen fest:

1. Viele Definitionen eines C-Programms werden in den sogenannten Include-Dateien festgelegt, beispielsweise Variablentypen, festgelegte Übergabeparameter und der Aufbau wichtiger Amiga-Strukturen. Hat man selber einen C-Compiler, kann man sich die Definitionen dort in den Include-Dateien anschauen, ansonsten findet man die wichtigsten beispielsweise im Amiga Intern.

2. Eine wichtige Funktion in C haben die Strukturen, die in etwa einer Tabelle entsprechen. Im Gegensatz zu Assembler haben dort aber alle Einträge automatisch einen Namen und durch den Variablentyp auch eine feste Länge. Benötigt man nun den Offset eines Tabelleneintrags, muß man entweder die einzelnen Variablengrößen aufaddieren, oder man schreibt in der Tabelle direkt ein Label vor den Tabelleneintrag, oder man schaut im Amiga Intern nach, wo jeweils auch die Offsets angegeben sind.

3. Im Gegensatz zu einer Tabelle in Assembler wird durch die Definition einer Struktur nicht unbedingt auch der Platz im Speicher geschaffen, dazu dient meist eine spezielle Zeile, die eine neue Struktur im C-Programm mit den bestehenden Definitionen erzeugt. Beispielsweise erzeugt folgender Programmteil eine im Speicher abgelegte Struktur mit dem Namen NeuesFenster auf der Basis der Definition der Struktur NewWindow:

```
struct NewWindow NeuesFenster =
{
10,
10,
300,
100,
3,
... (und so weiter)
```

4. Beim Aufruf von Bibliotheksroutinen in C werden die Parameter hinter dem Namen des Unterprogramms in Klammern angegeben. Der C-Compiler sorgt dann (unsichtbar) dafür, daß die Parameter in die richtigen Register geschrieben werden. Während wir in Assembler den Rückgabewert in D0 erhalten und in einer Speicherstelle ablegen müssen, wird er in C direkt durch ein Gleichheitszeichen einer Variablen zugewiesen.

5. Der Zugriff in C auf ein Element einer Struktur kann auf zwei Weisen erfolgen. Etwas vereinfacht dargestellt: Ist die Struktur im Programm abgelegt worden, wird der Name, ein Punkt und dann der Name des Elements angegeben. Wird auf eine Struktur zugegriffen, deren Adresse von einer Unterroutine zurückgeliefert wird (beispielsweise die Fenstertabelle bei OpenWindow), dann benutzt man in C den Variablennamen, einen Pfeil und den Namen des Tabelleneintrags.

9.3 Stopp - Spiele auf Tastendruck anhalten

In diesem und dem folgenden Kapitel wollen wir Ihnen an zwei Beispielen zeigen, wie man Programmideeen umsetzt und sich die dafür notwendigen Informationen beschafft. Als Informationsquellen werden wir dabei im wesentlichen die beiden Intern-Bände verwenden. Beginnen wir mit dem ersten Beispiel.

Programmidee

Der Amiga hat ein Multitasking-Betriebssystem, es können also mehrere Aufgaben parallel abgearbeitet werden. Wenn sich nun aber Spiele einigermaßen an diese Spielregeln halten und das Betriebssystem nicht völlig ausschalten, müßte man doch so ein Spiel auf Tastendruck anhalten und mit einer anderen Taste wieder fortsetzen können. Bei vielen Spielen wird nämlich leider eine Pausenfunktion vergessen, so daß man nicht mal eben unterbrechen kann.

Unser Programm müßte also folgende grundsätzliche Fähigkeiten haben:

1. Die Priorität für den eigenen Task auf den größtmöglichen Wert setzen.

2. So lange warten, bis eine Stopp-Taste betätigt ist.

3. So lange mit der hohen Priorität alle Rechnerzeit verbrauchen, bis eine Weiter-Taste betätigt wird.

4. Außerdem sollte unser Programm mit einer speziellen Taste aus dem Speicher entfernbar sein, damit der Speicher wieder frei ist und die Stopp-Taste für andere Zwecke verwendet werden kann.

Als Stopp-Taste wollen wir die linke Shift-Taste verwenden, als Weiter-Taste die rechte Shift-Taste und als Abbruch-Taste die Ctrl-Taste.

Priorität auf den größtmöglichen Wert setzen

Kommen wir zur ersten Aufgabe unseres Programms: Wie können wir die Priorität eines Tasks verändern? Wir schauen im Amiga Intern bei den Task-Funktionen nach und finden das Unterprogramm SetTaskPri der Exec.library. Aus der Beschreibung entnehmen wir, daß in D0 die neue Priorität und in A1 ein Zeiger auf die Task-Struktur übergeben werden muß.

Bleibt also die Frage: Wie erhalten wir einen Zeiger (also die Adresse) unserer eigenen Task-Struktur. Wir könnten nun beispielsweise in der Execbase die gesamte Liste aller Tasks durchsuchen, um unseren eigenen Task zu finden, aber vorher schauen wir uns lieber noch die anderen verfügbaren Unterprogramme der Exec.library zu den Tasks an.

Tatsächlich finden wir ein Unterprogramm FindTask, das einen beliebigen Task finden kann. Dazu muß nur der Name des Task angegeben werden. Da unser Programm ja Stop heißen soll, könnten wir natürlich nach einem Task mit dem Namen Stop suchen lassen. Aber das könnte schon deshalb schiefgehen, weil jemand unser Programm ja umbenennen könnte. Bei der Beschreibung der Unterroutine FindTask finden wir aber einen äußerst nützlichen Hinweis: Gibt man statt des Namens eine Null an, erhält man die Adresse der Task-Struktur des eigenen Task. Damit haben wir alle notwendigen Informationen für die Aufgabe zusammen.

Auf die Stopp-Taste warten

Relativ einfach ist die Aufgabe, auf die Stopp-Taste zu warten. Wir müssen nur daran denken, daß unser Task mit der größtmöglichen Priorität arbeitet. Wenn wir beispielsweise in einer Schleife direkt eine Speicherstelle abfragen würden, bekämen alle anderen Tasks überhaupt keine Zeit mehr - der Amiga wäre gestoppt, obwohl die Stopp-Taste noch gar nicht betätigt wurde.

Allerdings haben wir ja schon die Unterroutine Delay der Dos.library kennengelernt, die unseren Task für die angegebene Zeit "auf Eis legt", so daß in dieser Zeit auch andere Tasks mit

einer geringeren Priorität Prozessorzeit erhalten. Dieser Programmteil wird also etwa folgendermaßen aussehen:

- Mit Delay warten
- Stopp-Taste gedrückt ?
- Nein, dann zurück zur Warteschleife
- Ja, alle Prozessorzeit verbrauchen und auf Weiter-Taste achten

Bleibt nur die Frage, wie lange wir denn jeweils warten wollen. Wenn wir eine zu kurze Zeit warten, wird der Amiga spürbar langsamer werden, wenn wir eine zu lange Zeit warten, muß die Stopp-Taste zu lange gedrückt werden, bis unser Programm den Tastendruck merkt. Wir entscheiden uns für 1/5 Sekunde und müssen daher der Unterroutine Delay den Wert 10 (für 10/50 Sekunden) übergeben.

Tasks stoppen und auf Weiter-Taste warten

Im Gegensatz zum Warten auf die Stopp-Taste müssen wir beim Stopppen und Warten auf Weiter-Taste die gesamte Rechenzeit beanspruchen. Dadurch erhalten die anderen Tasks und damit auch das Spiel keine Rechenzeit mehr. Allerdings dürfen wir nun nicht etwa eine Betriebssystem-Unterroutine aufrufen, um etwa einen Tastendruck zu erfragen. Denn wir müssen damit rechnen, daß kein anderer Task Rechenzeit erhält und folglich auch keine Tastaturabfrage durch das Betriebssystem stattfinden kann.

An welcher Stelle können wir denn nun direkt die Hardware der Tastatur abfragen? Im Kapitel "Die Datenübertragung" des Intern erfahren wir, daß die Tastatur mit CIA A verbunden ist. Wir schlagen in Kapitel "Das CIA 8520" nach und erhalten den Hinweis, daß man mit dem Register ICR die Daten lesen kann. Etwas weiter unten finden wir dann auch die Adresse des ICR-Registers der CIA-A: $BFEC01.

Nun müssen wir noch herausfinden, welchen Wert dieses Register (also die Speicherstelle) annimmt, wenn eine unserer drei

speziellen Tasten betätigt wird. Wir schreiben dazu ein kleines Test-Programm, das uns den ersten Tastendruck als Zahl liefert. Als Grundprinzip verwenden wir die Ideen, mit der wir in einem der ersten Kapitel Mausbewegungen festgestellt haben: Wir merken uns einen Startwert und vergleichen von dem Augenblick den aktuellen Wert mit dem Startwert. Sobald diese nicht mehr identisch sind, beenden wir das Programm und haben den Wert der gedrückten Taste:

```
Tasten      = $BFEC01

Start:
MOVE.B Tasten,D1
CLR.L D0
Schleife:
MOVE.B Tasten,D0
CMP.B D0,D1
BEQ Schleife
ILLEGAL
```

Nach der Eingabe des Programms können Sie nach dem Start für jede Taste den zugehörigen Wert im Register ermitteln.

Übrigens werden Sie beim Erstellen und Nutzen des Programms feststellen, daß man doch recht gerne wissen möchte, wann das Programm in der Stopp-Schleife ist. Folglich werden wir innerhalb dieser Stopp-Schleife noch die LED blinken lassen. Die notwendige Adresse der LED und das Blinken haben wir ja schon in den ersten Beispielprogrammen kennengelernt.

Bevor wir das fertige Programm abdrucken, möchten wir Sie doch zumindest auffordern, es selbst zu versuchen. Denn die nötigen Kenntnisse haben Sie inzwischen und am meisten lernt man, wenn man selbst programmiert. In jedem Fall finden Sie - wenn es Probleme gibt oder Sie nur einfach Ihre mit unserer Lösung vergleichen wollen - das fertige Programm.

```
Das Programm STOP.S

; Definitionen

OpenLibrary = -$228
FindTask    = -$126
SetTaskPri  = -$12C
```

```
Delay        = -$0C6

Tasten       = $BFEC01
LED          = $BFE001
Shiftlinks   = 63
Shiftrechts  = 61
Control      = 57
Maxtaskpri   = 127        ; maximal mögliche Priorität
Anzticks     = 10         ; 10/50 Sekunden Warten (für Delay)
WartBlink    = 10000      ; Zähler für LED-Blinken

Start:
Nop
JSR Init               ; Libraries öffnen
BEQ Ende               ; Fehler
JSR InitTask           ; Task vorbereiten
Beq Ende               ; Fehler, Ende

Schleife:
MOVE.l Dosbase,A6      ;
Move.l #Anzticks,D1    ; Wartezeit setzen
JSR Delay(A6)          ; Warten, damit andere Tasks drankommen

CMP.B #Control,Tasten    ; Control-Taste gedrückt ?
BEQ Ende                 ; ja, Ende
CMP.B #Shiftlinks,Tasten ; Sollen wir alles anhalten?
BNE Schleife             ; Nein, von vorne

; Wir sollen alles anhalten...
Stopall:
  MOVE.L #Wartblink,D0   ; Zähler für LED-Blinken setzen
  BCHG #1,LED            ; LED umschalten
Warte:
    SUB.L #1,D0          ; Zähler verringern
    BNE Warte1           ; nicht 0, dann weiter
    BCHG #1,LED          ; Zähler 0, also LED umkehren -> blinken
    MOVE.L #Wartblink,D0 ; Zähler neu setzen
Warte1:
    CMP.B #Shiftrechts,Tasten   ; Sollen wir weitermachen ?
    BNE Warte                   ; Nein, noch gestoppt lassen
Warteend:
MOVE.B #00,LED           ; Es geht weiter, LED wieder einschalten
Jmp Schleife             ; Zurück zur Hauptschleife

Ende:
MOVE.L MyTask,A1         ; Unsere Taskadresse holen
CMP #00,A1               ; Ist die vielleicht gar nicht da?
BEQ Ende1                ; Keine Taskadresse -> Ende
MOVE.L Execbase,a6
MOVE.L #0,D0             ; Priorität auf 0 = Standard
JSR SetTaskPri(A6)       ; Taskpriorität setzen
MOVE.L #0,D0             ; Rückgabewert an DOS

Ende1:
```

```
RTS                         ; Zurück zum DOS

; Task finden und Priorität hochsetzen
InitTask:
MOVE.L #0,A1              ; Statt Name, eigenen Task ermitteln
MOVE.L Execbase,a6
JSR FindTask(A6)          ; Unsere Taskadresse bestimmen
MOVE.L D0,MyTask          ; und merken
TST.L D0                  ; gab es Probleme?
BEQ EndInitTask           ; Task nicht gefunden, Ende

Move.l #Maxtaskpri,d0     ; Maximale Priorität für
MOVE.l MyTask,A1             ; unseren Task
MOVE.L Execbase,A6
JSR SetTaskPri(A6)        ; Maximale Priorität für uns setzen
MOVE.l #1,D0              ; Alles klar an Hauptroutine melden
TST.L D0                  ; Und geprüft zurückliefern
EndInitTask:
RTS

; Hole Exec- und Dos-Library
Init:
MOVE.L 4,Execbase          ; Exec-Adresse holen
MOVE.L 4,A6
MOVE.L #0,D0               ; Version ist gleichgültig
LEA Dosname,A1             ; "dos.library"
JSR OpenLibrary(A6)        ; Dos.Library öffnen
MOVE.L D0,Dosbase          ; Startadresse der Library merken
TST.L D0                   ; Gab es Probleme mit der doslibrary?
RTS

; Variablen

Even
Execbase: dc.l 0
Dosbase:  dc.l 0
MyTask:   dc.l 0
Dosname:  dc.b `dos.library`,0
```

9.4 Compare - Dateien vergleichen

Im folgenden Programm kommen eine ganze Menge schon bekannter Vorgänge rund um Dateien vor. Wir haben uns dieses Programm aus zwei Gründen überlegt:

1. Weil unserer Meinung nach ein entsprechender CLI-Befehl fehlt.

2. Weil wir den Mechanismus kennenlernen wollen, mit dem man Parameter direkt aus der Kommandozeile des CLI entnehmen kann und nicht nachträglich erfragen muß.

Programmidee

Manchmal möchte man wissen, ob zwei Dateien (zum Beispiel Texte oder Programme) identisch sind. Da ein entsprechender CLI-Befehl COMP (für compare = vergleichen) fehlt, wollen wir uns einfach selbst einen schreiben.

Funktionsweise des Programms

Die grundsätzlichen Programmfunktionen sind alle bekannt. Die einzelnen Schritte sind folgende.

1. Die beiden zu vergleichenden Dateien ermitteln.

2. Die beiden Dateien öffnen.

3. Aus beiden Dateien jeweils ein Zeichen lesen und vergleichen.

4. Die beiden Dateien sind gleich, wenn jeweils die beiden Zeichen gleich sind und mit dem Ende der ersten Datei auch die zweite Datei endet.

5. Die beiden Dateien sind nicht identisch, wenn irgendwann einmal die beiden zu vergleichenden Zeichen nicht übereinstimmen oder eine der beiden Dateien länger als die andere ist.

So erhält man übergebene Parameter aus der Kommandozeile

Eine Besonderheit, die auch alle CLI-Befehle besitzen, wollen wir allerdings in das Programm einbauen: Die Parameter werden nicht nachträglich erfragt, sondern schon beim Programmaufruf hinter dem Programmnamen angegeben.

Da das Programm vom CLI aus gestartet werden soll, schauen wir im Intern nach, wie Programme von Amiga-DOS gestartet werden und wo man die Parameter finden kann. Das Ergebnis finden wir in Kapitel "Programme". Beim Start erhält ein Programm im Register A0 die Adresse der Parameter und in D0 die Anzahl der Zeichen der Parameter. Im Kapitel steht auch gleich ein einfaches Programm, das einen Parameter holt und verarbeitet.

Damit haben wir alle notwendigen Informationen für unser Programm, müssen allerdings noch zwei Besonderheiten beachten:

- Selbst wenn kein Zeichen als Parameter angegeben wurde, erhalten wir in D0 mindestens eine 1, weil auch die Return-Taste (Wert $0A = 10) als Angabe zählt.

- Die Parameter werden durch mindestens ein Leerzeichen getrennt. Es können aber auch beliebig viele Leerzeichen angegeben werden. Dies sollte unser Programm berücksichtigen und die Leerzeichen aus der Parameterzeile entfernen.

Das folgende Programm sollten Sie natürlich wieder zuerst selbst ausprobieren, bevor Sie sich unsere Lösung anschauen. Wir haben bei unserem Programm darauf Wert gelegt, möglichst viele Fehlermöglichkeiten zu berücksichtigen und auf dem Bildschirm zu melden. Das CLI-Fenster zur Ausgabe von Meldungen holen wir uns übrigens diesmal nicht durch das Öffnen eines Fensters mit dem Namen "*", sondern über eine neue Unterroutine der Dos.library: Output. Das hat den Vorteil, daß unser Programm COMP die Meldungen beispielsweise auch in eine Datei ausgeben kann, wenn die Ausgabe der CLI-Befehle mit dem Umlenkzeichen ">" in eine Datei umgelenkt wurde. Weitere Informationen dazu finden Sie im Amiga Intern beim entsprechenden Unterprogramm der Dos.library.

Ein letzter Hinweis: Unser Programm bricht mit einer entsprechenden Meldung ab, sobald die Dateien unterschiedlich sind. Sie können es dahingehend erweitern, daß beispielsweise die Unterschiede auf dem Bildschirm ausgegeben werden.

```
; Das Programm COMP.S zum Vergleichen von Dateien

; Definitionen
Execbase     =  4
Mode_Oldfile = 1005        ; Datei-Modus: bestehende Datei
OpenLibrary  = -$228
CloseLibrary = -$19E
Delay        = -$0C6
OUTPUT       = -$03C
OPEN         = -$01E
CLOSE        = -$024
READ         = -$02A
WRITE        = -$030

Falsch  = 0                ; Rückgabewert von Unterprogrammen
Wahr    = 1                ; Rückgabewert von Unterprogrammen

Start:
Nop
MOVE.B D0,Paramlen       ; Anzahl Zeichen der Parameter sichern
MOVE.L A0,Paramadress    ; Adresse der Parameter sichern
JSR Init                 ; Libraries öffnen
BEQ Ende1                ; Fehler
JSR GetOutput            ; aktuelle Ausgabe holen (für Meldungen)
BEQ Ende                 ; Fehler
JSR GetFiles             ; Dateinamen aus Kommandozeile holen
CMP.L #Falsch,D0
Beq Ende                 ; Fehler, Ende
JSR Openfiles            ; Dateien öffnen
CMP.L #Falsch,D0
BEQ CloseFiles           ; eine oder beide Dateien nicht geöffnet
JSR Vergleiche           ; hier werden die Dateien gelesen und
                           verglichen

; Es werden nur geöffnete Dateien geschlossen!!!
Closefiles:
MOVE.L Dosbase,A6
Move.L Handle1,D1
TST.L D1                 ; Ist Datei 1 überhaupt geöffnet
BEQ Closefiles1          ; Nein, also auch nicht schließen
JSR CLOSE(A6)
Closefiles1:
Move.L Handle2,D1
TST.L D1                 ; Ist Datei 1 überhaupt geöffnet
BEQ Ende                 ; Nein, also auch nicht schließen
JSR CLOSE(A6)

; Dos.library nur schließen, wenn auch geöffnet
Ende:
MOVE.L Execbase,a6         ; Basis Exec.Library
MOVE.L Dosbase,A1          ; Basisadresse Dos.Library
CMP.L #0,A1                ; Ist dies eine gültige Basisadresse?
BEQ Ende1                  ; nein, nicht schließen
JSR CloseLibrary(A6)       ; Dos.Library schließen
```

```
Ende1:
RTS                        ; Zurück zum DOS


; Hole Exec- und Dos-Library
Init:
MOVE.L Execbase,A6        ; Basisadresse der Exec.library
MOVE.L #0,D0              ; Version ist gleichgültig
LEA Dosname,A1            ; "dos.library"
JSR OpenLibrary(A6)       ; Dos.Library öffnen
MOVE.L D0,Dosbase         ; Startadresse der Library merken
TST.L D0                  ; Gab es Probleme mit der Dos.library?
RTS

GetOutput:
MOVE.L Dosbase,A6         ;
JSR Output(A6)            ; Aktuelles Ausgabegerät holen
MOVE.L D0,Ausgabe         ; und merken
TST.L D0
RTS

; Dieses Unterprogramm gibt einen Text aus
; Adresse und Länge müssen in D2 und D3 gesetzt sein!!!
Print:
MOVE.L Dosbase,A6
MOVE.L Ausgabe,D1
JSR Write(A6)
TST.L D0
RTS

OpenFiles:
MOVE.L Dosbase,A6
MOVE.L #Datei1,D2           ; Dateiname 1 ausgeben
CLR.L D3
MOVE.B Datlen1,D3
JSR Print
MOVE.L #Datei1,D1           ; Datei 1 öffnen
MOVE.L #Mode_Oldfile,D2     ; Modus: alt
JSR OPEN(A6)
MOVE.L D0,Handle1
TST.L D0
BEQ OpenFileFehler          ; gibt Fehlermeldung und Ende
MOVE.L #Textopen,D2         ; gibt Bestätigung auf Bildschirm
MOVE.L #Textopenende-Textopen,D3
JSR Print

MOVE.L Dosbase,A6
MOVE.L #Datei2,D2
CLR.L D3
MOVE.B Datlen2,D3
JSR Print
MOVE.L #Datei2,D1
MOVE.L #Mode_Oldfile,D2     ; Mode: alt
JSR OPEN(A6)
```

```
MOVE.L D0,Handle2
TST.L D0
BEQ OpenFileFehler
MOVE.L #Textopen,D2
MOVE.L #Textopenende-Textopen,D3
JSR Print

MOVE.L #Wahr,D0
RTS

OpenFileFehler:
MOVE.L #TextopenFehler,D2
MOVE.L #TextopenFehlerende-TextopenFehler,D3
JSR Print
MOVE.L #Falsch,D0
RTS

Vergleiche:
MOVE.L DOSBASE,A6
MOVE.l Handle1,D1
MOVE.L #Puffer1,D2
MOVE.L #01,D3
JSR Read(A6)              ; 1 Byte aus Datei 1 in Puffer 1
CMP.L #01,D0              ; kein Zeichen mehr ?
BNE File1End              ; nein, 2. Datei prüfen
MOVE.l Handle2,D1
MOVE.L #Puffer2,D2
MOVE.L #01,D3
JSR Read(A6)              ; 1 Byte aus Datei 2 in Puffer 2
CMP.L #01,D0
BNE VergleicheFehler      ; Datei 2 kürzer
MOVE.B Puffer1,D0
CMP.B Puffer2,D0
BNE VergleicheFehler      ; Dateien sind ungleich
JMP Vergleiche            ; und weiter lesen

; Erste Datei zu Ende, zweite muß auch zu Ende sein
File1End:
MOVE.l Handle2,D1
MOVE.L #Puffer2,D2
MOVE.L #01,D3
JSR Read(A6)
CMP.L #0,D0
BNE VergleicheFehler      ; Datei 1 zu Ende, Datei 2 nicht

; Dateien sind gleich, entsprechenden Text ausgeben
Vergleicheok:
MOVE.L #Textgleich,D2
MOVE.L #Textgleichende-Textgleich,D3
JSR Print
MOVE.L #Wahr,D0
RTS

; Dateien sind nicht gleich, entsprechenden Text ausgeben
```

```
VergleicheFehler:
MOVE.L #Textungleich,D2
MOVE.L #Textungleichende-Textungleich,D3
JSR Print
MOVE.L #Falsch,D0
RTS

; Zwei Dateinamen aus Kommandozeile holen und Länge merken
GetFiles:
CLR.L D0
MOVE.B Paramlen, D0       ; Länge der Parameter holen
MOVE.L Paramadress,A0     ; Adresse der Parameter
LEA Datei1,A1             ; Puffer für ersten Dateinamen holen

RemBlank1:                ; Leerzeichen entfernen
CMP.B #10,(A0)            ; Ist Zeichen ein CR ?
BEQ GetFilesError         ; Ja, Fehler! keine 2 Dateinamen
CMP.B #32,(A0)            ; Ist Zeichen ein Leerzeichen ?
BNE SchleifeGf1           ; nein, ersten Dateinamen holen
ADD.L #01,A0              ; Ein Zeichen weiterrücken
SUB.L #01,D0              ; Ein gültiges Zeichen weniger
BEQ GetFilesError         ; 0 ? -> keine zwei Dateinamen
JMP RemBlank1

SchleifeGF1:
TST.L D0                   ; sind überhaupt noch Zeichen da?
BEQ GetFilesError          ; nein, schade -> Fehler
MOVE.B (A0)+,D1            ; Zeichen in Register holen
CMP.B #10,D1               ; Ist Zeichen ein CR?
BEQ GetFilesError          ; Ja, zweiter Name fehlt -> Fehler
CMP.B #32,D1               ; Leerzeichen ?
BEQ StartGF2               ; dann folgt zweiter Name
MOVE.B D1,(A1)+            ; Zeichen ablegen
SUB.L #01,D0               ; Ein Zeichen weniger
ADD.B #01,Datlen1          ; Länge erhöhen
JMP SchleifeGF1            ; Zurück zur Schleife

StartGF2:                 ; Puffer abschließen und zweiten holen
MOVE.B #0,(A1)            ; Puffer mit 0 abschließen
LEA Datei2,A1             ; Puffer für zweiten Dateinamen holen

RemBlank2:                ; Leerzeichen entfernen
CMP.B #10,(A0)            ; Ist Zeichen ein CR ?
BEQ GetFilesError         ; Ja, Fehler
CMP.B #32,(A0)            ; Ist Zeichen ein Leerzeichen ?
BNE SchleifeGf2           ; nein, zweiten Dateinamen holen
ADD.L #01,A0              ; Ein Zeichen weiterrücken
SUB.L #01,D0              ; Ein gültiges Zeichen weniger
BEQ GetFilesError         ; 0 ? -> keine zwei Dateinamen
JMP RemBlank2

SchleifeGF2:
MOVE.B (A0)+,D1            ; Zeichen in Register holen
CMP.B #10,D1               ; Ist Zeichen ein CR ?
```

```
BEQ GetFilesEnde           ; Ja, Puffer abschließen und fertig
CMP.B #32,D1               ; Leerzeichen ?
BEQ GetFilesEnde           ; Leerzeichen beendet zweiten Namen
MOVE.B D1,(A1)+            ; Zeichen ablegen
ADD.B #01,Datlen2
SUB.L #01,D0               ; Ein Zeichen weniger
BNE SchleifeGF2            ; nicht fertig? -> Weitere Zeichen holen

GetFilesEnde:
MOVE.B #0,(A1)             ; zweiten Puffer mit 0 abschließen
MOVE.L #Wahr,D0
RTS

GetFilesError:
MOVE.L #Syntax,D2
MOVE.L #Syntaxende-Syntax,D3
JSR Print
MOVE.L #Falsch,D0
RTS

; Variablen

Even
Execbase: dc.l 0
Dosbase:  dc.l 0
Paramadress: dc.l 0
Ausgabe: dc.l 0
Handle1: dc.l 0
Handle2: dc.l 0

Dosname:  dc.b `dos.library`,0

Syntax: dc.b `Syntax: COMP Datei1 Datei2`,10
Syntaxende:

Textopen: dc.b ` geöffnet`,10
Textopenende:

TextopenFehler: dc.b ` kann nicht geöffnet werden`,10
TextopenFehlerende:

Textgleich: dc.b `Dateien sind gleich`,10
Textgleichende:

Textungleich: dc.b `Dateien sind unterschiedlich!!!`,10
Textungleichende:

Datei1:   blk.b 50,0
Datei2:   blk.b 50,0
Paramlen: dc.b 0
Datlen1: dc.b 0
```

```
Datlen2: dc.b 0
Puffer1: dc.b 0
Puffer2: dc.b 0
```

9.5 Was man noch wissen sollte

Wir nähern uns nun dem Ende des Buches und hoffen, daß Sie die Welt der Maschinensprache tatsächlich problemlos und mit viel Spaß kennengelernt haben. Zugegebenermaßen haben wir an einigen Stellen ganz schön vereinfacht und manchmal lieber auf "sauberen Programmierstil" verzichtet, wenn uns die Lösung einfacher schien oder wir dadurch auf die Einführung weiterer Befehle und Adressierungsarten verzichten konnten.

Daher wollen wir an dieser Stelle kurz auf einige Themen eingehen, von denen Sie zumindest wissen sollten.

Welche Register man verwenden kann

So lange Sie kleine Programme schreiben und das Betriebssystem des Amiga kaum verwenden, können Sie natürlich mit den Registern des Prozessors machen, was Sie wollen. Aber schon bei längeren Programmen mit vielen Unterprogrammen kann dies schnell zu Fehlern führen und auch beim Aufruf von Unterroutinen aus Amiga-Bibliotheken sollte man wissen, welche Register vom Betriebssystem geändert werden können und welche nicht.

Von den Entwicklern des Amiga ist die Nutzung der Register fest vorgeschrieben: Die Register D0-D1 und A0-A1 können beliebig verändert werden, die anderen Register bleiben auch beim Aufruf einer Betriebssystemfunktion unverändert. Wenn Sie also eine Betriebssystemfunktion verwenden wollen und in den Registern D0-D1 oder A0-A1 wichtige Daten haben, müssen Sie diese vor dem Aufruf der Routine speichern und anschließend wieder in diese Register schreiben.

Auch bei eigenen Unterprogammen sollte man sich, wenn möglich, an diese Regeln halten. Zugegebenermaßen haben wir an einigen Stellen eine Ausnahme gemacht, in dem wir beispiels-

weise in Unterprogrammen D2 und D3 verwendet haben, ohne diese zu sichern. Aber zu diesem Zeitpunkt kannten wir noch keinen komfortablen Mechanismus, um ein oder mehrere Register sichern zu können. Etwas weiter unten werden wir Ihnen einen solchen Mechanismus vorstellen.

Noch einmal an dieser Stelle wollen wir daran erinnern, daß das Register A7 keinesfalls zur normalen Bearbeitung von Zahlen und Adressen verwendet werden darf, da es im Stack auf die jeweils aktuelle Rücksprungadresse zeigt. Trotzdem kann man das Adreßregister A7 mit einem Trick ideal zum einfachen Sichern von Registerwerten verwenden.

So sichert man den Inhalt eines Registers - Nutzung des Stacks

Stellen Sie sich einmal vor, Sie haben 20 Unterprogramme und diese müssen jeweils ein oder zwei Register sichern. Wenn Sie nun jeweils entsprechende Label am Programmende setzen und dort mit dem Befehl DC.L den nötigen Platz schaffen, wird Ihr Programm ganz schön lang, und Arbeit macht das Ganze auch noch.

Es gibt eine einfache Möglichkeit, zur Sicherung eines Registers den Stack zu benutzen, also den Bereich des Speichers, in dem die Rücksprungadressen aufbewahrt werden. Dieser Speicher wird nämlich normalerweise nur zu einem kleinen Teil von diesen Rücksprungadressen belegt, der Rest ist frei.

Bevor wir die Funktionsweise genau erklären, wollen wir die Vorgehensweise an einem Beispiel zeigen:

```
MOVE.L D0,-(A7)          ; Register auf Stack sichern
; hier kann das Register geändert werden
; da es ja gerettet ist
MOVE.L (A7)+,D0          ; Register zurückholen
```

Hinweis: Beim SEKA MUSS statt MOVEM der Befehl MOVE verwendet werden!

Kommen wir nun zur Erklärung der Funktionsweise: Durch das Minuszeichen vor der runden Klammer wird der Wert des Regi-

sters geändert, bevor der Rest des Befehls abgearbeitet wird. Dadurch wird sozusagen der notwendige Platz auf dem Stack geschaffen. Anschließend wird der Inhalt von D0 an der Adresse im Speicher abgelegt, die in A7 steht. Die Adressierung mit den runden Klammern kennen Sie ja schon als "Indirekte Adressierung". Wieviel Platz auf dem Stack geschaffen wird, hängt von der Operandenlänge (in unserem Fall Langwort wegen ".L") ab.

Im Gegenstück zur ersten Befehlszeile wird der zuvor gesicherte Wert aus D0 aus dem Speicher zurück in das Datenregister geholt, dann wird durch das Pluszeichen wieder der alte Wert in A7 hergestellt. Übrigens sollte man noch wissen, daß der Stack immer von hohen Speicheradressen zu niedrigen mit Werten gefüllt wird, also sozusagen "nach unten wächst". Eine ganz schön komplizierte Angelegenheit, nicht wahr? Am besten machen wir uns den Vorgang an einem kleinen Schaubild klar. Zuerst die Situation vor unserem ersten Befehl:

```
Stack                         Adresse
 | benutzter                 |  992         Register A7
 | Stackbereich              |-
 |                           |  996        ----------------
 |                           |             |     1000     |
 |---------------------------|--           ----------------
 | aktueller Rücksprung      | 1000   <------------|
 |---------------------------|--
 |                           | 1004
 | freier Stackbereich       |-
 |                           | 1008
```

Das Register A7 (Stackpointer) zeigt also auf die Adresse 1000, die die aktuelle Rücksprungadresse enthält. Nach dem Befehl "MOVE.L D0,-(A7)" sieht die Situation folgendermaßen aus:

```
Stack                         Adresse
 | benutzter                 |  992         Register A7
 | Stackbereich              |-
 |                           |  996        ----------------
 |                           |             |     1004     |
 |---------------------------|--           ----------------
 | aktueller Rücksprung      | 1000                |
 |---------------------------|--                   |
 |             1             | 1004    <-----------|
 |---------------------------|
 |                           | 1008
 | freier Stackbereich       |-
 |                           | 1012
```

Nach der letzten Befehlszeile "MOVE.L (A7)+,D0" ist dann wieder die Ausgangssituation hergestellt.

Hinweis: Außerordentlich wichtig ist, daß ein gesichertes Register auch wieder "vom Stack geholt" wird, denn geschieht dies nicht und erfolgt irgendwann ein Rücksprung mit RTS, findet der Prozessor nicht die richtige Rücksprungadresse, sondern die auf dem Stack gespeicherte 1, und es kommt zum Programmabsturz.

Mehrere Register mit einem Befehl sichern

Um mehrere Register mit einem Befehl auf dem Stack abzulegen und später wieder zurückzuholen, gibt es eine spezielle Variante des Befehls MOVE: den Befehl MOVEM. Dahinter wird eine Liste der zu sichernden Register angegeben. Aufeinanderfolgende Register werden mit Bindestrich angegeben (D0-D3 sichert D0,D1,D2,D3) und einzelne Register werden durch den Schrägstrich getrennt (D0/D1/A0 sichert D0, D1 und A0). Das folgende Beispiel sichert alle Register auf dem Stack:

```
MOVEM.L D0-D7/A0-A6,-(A7) ; Alle Register auf Stack sichern
; hier können alle Register geändert werden
MOVEM.L (A7)+,D0-D7/A0-A6 ; Alle Register zurückholen
```

Hinweis: Das Register A7 kann natürlich selbst nicht gerettet werden, da es sich ja während des Befehls ändert.

Mit diesem Wissen können wir nun zu Beginn eines Unterprogramms alle notwendigen Register sichern und vor dem Rücksprung mit RTS wieder zurückholen.

Besonderheiten bei negativen Zahlen

Zugegebenermaßen haben wir beim Thema negative Zahlen einen zwar nicht ganz korrekten, aber unserer Meinung nach dem Einsteiger gerechten Weg eingeschlagen. Wir haben sie einfach benutzt, als wären sie ganz natürlich und ansonsten kein

Wort der Erklärung verloren. Und anscheinend hat ja auch alles recht gut funktioniert. An dieser Stelle wollen wir die Erklärung nun kurz nachholen.

Mit einem Byte können 256 verschiedene Zahlen von 0 bis 255 dargestellt werden. Wie kommt man aber zu den negativen Zahlen? Die Lösung ist einfach: Man nimmt einfach das höchstwertige Bit (also das Bit 7) und definiert es als Vorzeichenbit. Ist dieses Bit 0, ist die Zahl positiv, sonst negativ. Dadurch können mit einem Byte nicht mehr die Zahlen von 0-255 sondern jetzt von -128 bis +127 dargestellt werden. Mit einem Byte läßt sich die Zahl +128 nicht mehr darstellen. Übrigens sind beide Zahlendefinition völlig gleichwertig, es ist nur eine Frage der Sicht- und Rechenweise:

Ohne Vorzeichen:	$00	$01	$7F	$80	$81	$FE	$FF
Mit Vorzeichen :	$00	$01	$7F	-$80	-$7F	-$02	-$01

Wenn Sie im Assembler eine negative Zahl verwenden, wird diese automatisch in ihr positives Gegenstück umgewandelt. Der SEKA wandelt also -$01 automatisch in $FF um. Wichtig ist nur, daß Sie je nach Zahlensystem die richtigen Befehle und Rechenvorschriften verwenden. So gibt es beispielsweise zwei Multiplikationsbefehle: MULS multipliziert zwei Zahlen mit Vorzeichen (das "S" kommt von signed = mit Vorzeichen), MULU zwei Zahlen ohne Vorzeichen. Dieser Unterschied ist ja auch logisch, denn im ersten Fall ist das höchste Bit nur das Vorzeichen (Rechenregel: Minus mal Minus ist Plus ...), ansonsten eine Ziffer der Zahl im binären Zahlensystem, die multipliziert werden muß.

Ganz so schlimm ist das aber nicht mit den negativen Zahlen, denn - wie gesagt - nimmt einem der SEKA in den meisten Fällen die Arbeit ab.

10. Anhang

In diesem Anhang wollen wir Ihnen Informationen zu den wichtigsten Befehlen des 68000 und den beiden Assemblern SEKA und Profimat zusammenstellen.

10.1 Befehlsübersicht

Der 68000 kennt eine ganze Reihe von Befehlen, von denen wir Ihnen hier die wichtigsten kurz erklären wollen.

ADD — **Addition**

Funktion: Addiert Quell- und Zieloperanden. Das Ergebnis wird in den Zieloperanden geschrieben.

Beispiel: `ADD.B #5,D0`

AND — **Logisches UND**

Funktion: Es erfolgt eine bitweise Verknüpfung von Quell- und Zieloperand, das Ergebnis wird ins Ziel gegeben. Die Verknüpfung liefert nur dann eine 1, wenn die Bits von Quell- und Zieloperand beide 1 sind. Ansonsten wird immer eine 0 zurückgeliefert.

Beispiel: `AND.B D0,$BFE001`

ASL — **Verschieben nach links**

Funktion: Der Zieloperand wird um die angegebene Stellenzahl nach links verschoben. Dabei werden in die niederwertigen Bits Nullen nachgeschoben.

Beispiel: `ASL.W #2,D0`

ASR — Verschieben nach rechts

Funktion: Der Zieloperand wird um die angegebene Stellenzahl nach rechts verschoben. Das höchstwertige Bit bleibt dabei erhalten, da es sich bei diesem Bit um das Vorzeichenbit handelt und dieses nicht verändert werden soll.

Beispiel: `ASR.W #3,D0`

BCHG — Bits ändern

Funktion: Prüft ein Bit und ändert es. Die Nummer des zu änderden Bits ist Quelloperand, die Speicherstelle in der sich das Bit befindet ist Zieloperand. Je nachdem, wie das Bit vor der Änderung gesetzt ist, wird auch das Zeroflag gesetzt, und zwar auf 1, falls das Bit gleich null ist und umgekehrt. Danach wird das Bit invertiert, also von 1 auf null oder von null auf 1 gesetzt.

Beispiel: `BCHG #01,$BFE001`

BEQ — Relativer Sprung, wenn Null

Funktion: Prüft, ob das Ergebnis der vorhergehenden Operation gleich null ist. Wenn ja, wird zum angegebenen Label verzweigt, sonst wird der nächste Befehl bearbeitet.

Beispiel: `BEQ ENDE`

BGE — Relativer Sprung, wenn größer oder gleich

Funktion: Prüft, ob das Ergebnis der vorhergehenden Operation größer oder gleich null ist. Wenn ja, wird zum angegebenen Label verzweigt, sonst wird der nächste Befehl bearbeitet.

Beispiel: `BGE TEST`

BGT **Relativer Sprung, wenn größer**

Funktion: Prüft, ob das Ergebnis der vorhergehenden Operation größer null ist. Wenn ja, wird zum angegebenen Label verzweigt, sonst wird der nächste Befehl bearbeitet.

Beispiel: `BGT SCHLEIFE`

BLE **Relativer Sprung, wenn kleiner oder gleich**

Funktion: Prüft, ob das Ergebnis der vorhergehenden Operation kleiner oder gleich null ist. Wenn ja, wird zum angegebenen Label verzweigt, sonst wird der nächste Befehl bearbeitet.

Beispiel: `BLE DOSBASE`

BLT **Relativer Sprung, wenn kleiner**

Funktion: Prüft, ob das Ergebnis der vorhergehenden Operation kleiner null ist. Wenn ja, wird zum angegebenen Label verzweigt, sonst wird der nächste Befehl bearbeitet.

Beispiel: `BLT WARTE`

BNE **Relativer Sprung, wenn nicht null**

Funktion: Prüft, ob das Ergebnis der vorhergehenden Operation ungleich null ist. Wenn ja, wird zum angegebenen Label verzweigt, sonst wird der nächste Befehl bearbeitet.

Beispiel: `BNE ENDE`

BRA **Relativer Sprung, verzweigt unbedingt**

Funktion: Sprung zum dahinter angegebenen Label. Im Gegensatz zu JMP wird nicht zu einer Adresse, sondern

relativ zum augenblicklichen Programmzähler um eine Distanz gesprungen. Dadurch können so geschriebene Programme im Speicher beliebig verschoben werden.

Beispiel: `BRA TEST`

BSR **Relativer Sprung in Unterroutine, verzweigt unbedingt**

Funktion: Innerhalb eines Programms wird immer in ein Unterprogramm verzweigt und zwar relativ zum augenblicklichen Programmzähler um eine Distanz. Dadurch können so geschriebene Programme im Speicher verschoben werden. Der Rücksprung erfolgt durch RTS.

Beispiel: `BSR SCHLEIFE`

BTST **Bit prüfen**

Funktion: Es wird ein Bit überprüft. Dabei steht die Nummer des zu prüfenden Bits im Quelloperanden, die Speicherstelle, in der sich das zu prüfende Bit befindet, steht im Zieloperanden. Das Ergebnis der Abfrage geht ins Zeroflag. Ist das zu prüfende Bit gesetzt, so wird das Zeroflag auf null gesetzt, ist das Bit nicht gesetzt, so wird das Zeroflag auf eins gesetzt.

Beispiel: `BTST #4,D2`

CLR **Inhalt eines Operanden löschen**

Funktion: Setzt ein Datenregister oder eine Speicherstelle auf null (Operandenlänge .B .W .L).

Beispiel: `CLR.W D1`

CMP — Werte vergleichen

Funktion: Vergleicht zwei Operanden nach deren Größe. Dabei wird der Quell- vom Zieloperanden subtrahiert. Ist das Ergebnis gleich null, so wird das Zeroflag gesetzt, ist das Ergebnis ungleich null, so ist das Zeroflag gleich null. Der Inhalt von Quelle und Ziel bleibt trotz der Subtraktion unverändert.

Beispiel: `CMP.L D1,D2`

DIVS — Division mit Vorzeichen

Funktion: Dividiert den Zieloperanden (32 Bit) durch den Quelloperanden (16 Bit). Vorzeichen werden beachtet. Das Ergebnis wird in den Zieloperanden geschrieben. Dabei steht der Quotient (das Ergebnis) in den untersten 16 Bits, der Rest der Division in den obersten 16 Bits. Wichtig ist hier darauf zu achten, daß der Zieloperand nicht gleich null ist, da es sonst zu einem Fehler (Guru Meditation) kommt.

Beispiel: `DIVS #5,D3`

DIVU — Division ohne Vorzeichen

Funktion: Dividiert den Zieloperanden (32 Bit) durch den Quelloperanden (16 Bit). Vorzeichen werden nicht beachtet. Das Ergebnis wird in den Zieloperanden geschrieben. Dabei steht der Quotient (das Ergebnis) in den untersten 16 Bits, der Rest der Division in den obersten 16 Bits. Wichtig ist hier darauf zu achten, daß der Zieloperand nicht gleich null ist, da es sonst zu einer Fehlermeldung (Guru Meditation) kommt.

Beispiel: `DIVU #2,D4`

EOR — Logisches Exklusiv-ODER

Funktion: Es erfolgt eine bitweise Verknüpfung von Quell- und Zieloperand, das Ergebnis wird ins Ziel gegeben. Die Verknüpfung liefert dann eine 1, wenn entweder der Quell- oder der Zieloperand 1 sind. Sind beide Operanden gleich, wird immer eine 0 zurückgeliefert. Anders ausgedrückt werden im Ziel genau die Bits umgekehrt, die in der Quelle gesetzt sind.

Beispiel: `EOR.L %2,$BFE001`

ILLEGAL — Unzulässiger Maschinensprachebefehl

Funktion: Dieser Befehl führt zum Abbruch eines Programms, die Kontrolle wird wieder dem Assembler übergeben. Außerhalb eines Assemblers führt dieser Befehl zu einem "Task held..."-Requester.

Beispiel: `ILLEGAL`

JMP — Absoluter Sprungbefehl

Funktion: Innerhalb eines Programms wird immer zur festen Adresse eines Labels gesprungen. So geschriebene Programme können im Speicher nicht einfach verschoben werden.

Beispiel: `JMP ENDE`

JSR — Absoluter Sprung in ein Unterprogramm

Funktion: Innerhalb eines Programms wird immer in ein Unterprogramm verzweigt und zur festen Adresse eines Labels. So geschriebene Programme können im Speicher nicht einfach verschoben werden. Der Rücksprung erfolgt durch RTS.

Beispiel: `JSR WARTE`

LEA — Effektive Adresse ins Adressregister laden

Funktion: Die Adresse, nicht der Inhalt, des Quelloperanden wird in das angegebene Adreßregister geladen.

Beispiel: `LEA NAME,A0`

LSL — Logische Verschiebung nach links

Funktion: Der Zieloperand wird um die angegebene Stellenzahl nach links verschoben. Dabei werden in die niederwertigen Bits Nullen nachgeschoben.

Beispiel: `LSL.B #3,D4`

LSR — Logische Verschiebung nach rechts

Funktion: Der Zieloperand wird um die angegebene Stellenzahl nach rechts verschoben. In die höchstwertigen Bits wird dabei eine Null geschrieben.

Beispiel: `LSR.W 1,D2`

MOVE — Daten übertragen

Funktion: Überträgt Daten vom Quelloperanden in den Zieloperanden.

Beispiel: `MOVE.B $BFE001,D1`

MULS — Multiplikation mit Vorzeichen

Funktion: Multipliziert den Quelloperanden mit dem Zieloperanden. Vorzeichen werden beachtet. Das Ergebnis wird in den Zieloperanden geschrieben. Dabei werden nur die unteren 16 Bits des Quelloperanden mit

den unteren 16 Bits des Zieloperanden multipiziert. Das Ergebnis im Zieloperanden wird in die gesamten 32 Bits geschrieben.

Beispiel: `MULS #3,D1`

MULU **Multiplikation ohne Vorzeichen**

Funktion: Multipliziert den Quelloperanden mit dem Zieloperanden. Vorzeichen werden nicht beachtet. Das Ergebnis wird in den Zieloperanden geschrieben. Dabei werden nur die unteren 16 Bits des Quelloperanden mit den unteren 16 Bits des Zieloperanden multipiziert. Das Ergebnis im Zieloperanden wird in die gesamten 32 Bits geschrieben.

Beispiel: `MULS #2,D2`

NOP **Keine Operation**

Funktion: Bei diesem Befehl wird keine Operation ausgeführt, sondern nur der Programmzähler hochgezählt.

Beispiel: `NOP`

NOT **Logische Invertierung**

Funktion: Alle durch .B, .W oder. L angegebenen Bits des Operanden werden invertiert, also umgedreht. Das Ergebnis wird in den Operanden geschrieben.

Beispiel: `NOT.B D1`

OR **Logisches ODER**

Funktion: Es erfolgt eine bitweise Verknüpfung von Quell- und Zieloperand, das Ergebnis wird ins Ziel gegeben. Die Verknüpfung liefert nur dann eine 0, wenn

die Bits von Quell- und Zieloperand beide 0 sind. Ansonsten wird immer eine 1 zurückgeliefert.

Beispiel: `OR.B D1,$BFE001`

ROL **Rotieren nach links**

Funktion: Der Operand wird um den angegebenen Wert nach links rotiert. Dabei wird bei jedem Rotiervorgang jeweils das höchstwertige Bit in das niederwertige Bit geschoben.

Beispiel: `ROL.L #3,D4`

ROR **Rotieren nach rechts**

Funktion: Der Operand wird um den angegebenen Wert nach rechts rotiert. Dabei wird bei jedem Rotiervorgang jeweils das niederwertige Bit in das höchstwertigste Bit geschoben.

Beispiel: `ROR.L #2,D1`

RTS **Rücksprung aus einem Unterprogramm**

Funktion: Kehrt aus einem zuvor mit BSR oder JSR aufgerufenen Unterprogramm direkt hinter die Stelle zurück, von der aus das Unterprogramm aufgerufen wurde.

Beispiel: `RTS`

SUB **Subtraktion**

Funktion: Subtrahiert den Quell- vom Zieloperanden. Das Ergebnis wird in den Zieloperanden geschrieben.

Beispiel: `SUB.W #65,D3`

TST **Inhalt eines Operanden prüfen**

Funktion: Der Inhalt des Operanden wird daraufhin untersucht, ob er größer, kleiner oder gleich null ist. Je nach Ergebnis dieses Tests werden bestimmte Flags gesetzt. Ist der Inhalt des Operanden negativ, wird das Negativflag auf 1 gesetzt, ist er positiv wird das Zeroflag auf 0 gesetzt, ist der Inhalt gleich null wird das Zeroflag auf 1 gesetzt. Das Ergebnis kann anschließend mit einem Branch-Befehl abgefragt werden (z.B. BEQ, BNE usw.).

Beispiel: `TST.B D0`

10.2 Die Assembler

Wir haben in diesem Buch vorwiegend den SEKA-Assembler verwendet, weil er schnell und einfach zu bedienen ist. Damit Sie aber unsere Beispiele auch mit anderen Assemblern nachvollziehen können, fassen wir nicht nur die Befehle des SEKA zusammen, sondern zeigen auch die wichtigsten Unterschiede zu anderen Assemblern.

10.2.1 Der SEKA-Assembler

In diesem Abschnitt geht es um den SEKA von Kuma, den wir auch für die Programme in diesem Buch verwendet haben.

Vorteile des SEKA

Der SEKA gehört zu den Assemblern, die direkt im Speicher des Amiga assemblieren und daher auch sehr schnell sind. Außerdem ist der SEKA ein äußerst kurzes Programm, das nur sehr wenig Speicher im Amiga benötigt. Er eignet sich daher auch hervorragend für den Einsatz in einem Amiga mit 512 KByte Speicher.

Außerdem enthält der SEKA eine "komplette Entwicklungsumgebung", also vereinfacht gesagt alles, was man für die Programmierung in Assembler benötigt:

- Assembler (übersetzt Quelltext in Maschinensprache)
- Linker (kann mehrere Programmteile zu einem großen Programm zusammenfügen)
- Debugger (wird zur Fehlersuche in Programmen verwendet)
- Editor (zur Eingabe des Programms)

Interessant ist beim SEKA, daß dieser direkt ausführbare Programme im Speicher des Amiga erzeugen kann, oder Programme, die nach dem Speichern ohne den SEKA gestartet werden können, oder Objekt-Dateien, die später zu einem größeren Programm verbunden (gelinkt) werden können.

Grundsätzliche Arbeit mit dem SEKA

Rufen Sie den SEKA vom CLI aus mit seinem Programmnamen auf. Anschließend werden Sie nach dem Gesamtspeicher im Amiga gefragt, den Sie dem SEKA für die Arbeit zur Verfügung stellen wollen. Die Angabe erfolgt in KByte. Geben Sie beispielsweise 100 ein.

Nach dem Starten befindet sich der SEKA im Befehlsmodus. Mit der Taste <Esc> kann zwischen dem integrierten Editor und dem Befehlsmodus gewechselt werden. Alle Befehle werden im Befehlsmodus eingegeben.

Laden und Speichern von Programmen

Zum Speichern eines Programms benutzen Sie den Befehl W (von Write = Schreiben) und geben anschließend den Dateinamen gegebenenfalls mit Pfad an. Wenn Sie keine andere Erweiterung für den Dateinamen vorgeben, hängt der SEKA automatisch ein ".S" an. Zum Laden eines Programms benutzen Sie den Befehl R

(für Read = Lesen) und geben ebenfalls anschließend den Dateinamen an. Auch hier erwartet der SEKA, wenn nichts anderes angegeben, ist die Erweiterung ".S".

Hinweis: Der SEKA fügt mit R geladene Programme an der aktuellen Cursor-Position ein, löscht dadurch also nicht ein bestehendes Programm im Speicher. Dieses muß gegebenenfalls vorher mit KS (für Kill Source = Lösche Programmtext) und Bestätigung der Sicherheitsabfrage erfolgen.

Assemblieren und Starten

Um ein Programm im Speicher zu assemblieren und dort auch starten zu können, benutzen Sie den Befehl A und geben keine Optionen an, drücken also bei der Frage einfach die Return-Taste. Anschließend zeigt der SEKA fehlerhafte Zeilen mit einer Fehlermeldung an oder meldet "No errors", wenn kein Fehler aufgetreten ist.

Der Start erfolgt mit G (für Go), gegebenenfalls kann dahinter noch der Name eines Labels angegeben werden. Anschließend können mehrere Abbruchpunkte (Breakpoints) angegeben werden. Am besten verwendet man dazu auch die Namen von Labels, die man vorher im Programm definiert hat. Wird kein Breakpoint angegeben, sollte das Programm mit dem speziellen Befehl ILLEGAL enden, damit der SEKA dort wieder die Kontrolle erhält und die Register anzeigen kann. Das Programm kann auch mit einem RTS enden und wird dann mit J (für Jump) statt G gestartet.

Programme lauffähig speichern

Um ein Programm so zu erstellen und zu speichern, daß es ohne den SEKA direkt vom CLI aus gestartet werden kann, beenden Sie es mit einem RTS, assemblieren es ohne Angabe von Optionen und speichern es mit dem Befehl WO (für Write Objekt) ab. Der SEKA speichert das Programm unter dem anzugebenden Namen ohne Dateierweiterung ab.

Quickreferenz

Wenn Sie mit dem SEKA arbeiten, wird Ihnen diese Kurzübersicht bei der Arbeit mit diesem Buch helfen. Sie brauchen dann nicht immer im - ohnehin englischen - Handbuch nachzuschlagen.

Dateioperationen des SEKA

Die folgenden Kommandos dienen der Arbeit mit Dateien:

r Programmtext lesen

w Programmtext schreiben

wo Ausführbares Programm speichern

rl Linkfähige Datei laden

wl Linkfähige Datei speichern

ri Beliebige Datei in Speicher laden (von read image). Anschließend wird ein Dateiname, eine Start- und Endadresse erfragt. Um eine Datei komplett zu laden, als Endadresse -1 angeben.

wi Beliebige Speicherbereiche abspeichern. Anschließend einen Dateinamen, eine Start- und Endadresse angeben. Es können auch Label angegeben werden.

v Aktuelles Verzeichnis anzeigen. Wird ein Verzeichnisname hinter v angegeben, verwendet der SEKA dieses Verzeichnis von nun an zum Laden und Speichern und zeigt es zusätzlich an.

kf Datei löschen (kill file)

> Ausgabe in die anschließend angegebene Datei lenken. Kann durch erneute Angabe von ">" ohne Dateinamen beendet werden.

Der Editor des SEKA

Bei den folgenden Befehlen kann häufig eine Zeilennummer oder Anzahl (nr) angegeben werden. Entfällt diese, verwendet der SEKA eine Eins.

t nr (target nr): setzt den Cursor in die angegebene Zeile.

b (bottom): setzt den Cursor an das Textende.

u nr (up): bewegt den Cursor um die angegebene Anzahl Zeilen nach oben.

d nr (down): bewegt den Cursor um die angegebene Anzahl Zeilen nach unten.

z nr (zap): löscht die angegebene Anzahl Zeilen ab der aktuellen Cursor-Position.

e nr (edit): erlaubt das Editieren der aktuellen oder angegebenen Zeile.

ltext (locate): sucht den Text "text" ab der aktuellen Cursor-Position, zeigt die Zeile an und setzt den Cursor in die Zeile. Wird kein Suchtext angegeben, wird die Suche ab der aktuellen Cursor-Position mit dem letzten Suchtext wiederholt.

i (insert): Eingabemodus; es können nur noch Zeilen nacheinander eingegeben werden, Abbruch mit <Esc>.

ks (kill source): löscht nach einer Sicherheitsabfrage den Quelltext des Programms im Editor. Diesen Befehl sollten Sie verwenden, bevor Sie ein neues Programm anfangen oder laden.

o (old): hebt den Befehl ks wieder auf und holt damit den Text zurück. Muß sofort hinter versehentlichem ks eingegeben werden.

p nr (print): gibt die angegebene Anzahl Zeilen auf dem Bildschirm aus.

H (how big is): liefert Informationen über die augenblickliche Nutzung des Speichers, den Sie dem SEKA beim Starten

zur Verfügung gestellt haben. Die einzelnen Angaben werden nur angezeigt, wenn sie nicht Null sind, und haben folgende Bedeutung:

Work	gesamter verfügbarer Speicher für SEKA
Src	Programmtext
RelC	Relocation Information Code
RelD	Relocation Information Data
Code	Objekt Code
Data	Objekt Data
Link	Größe des Link-Pufferbereichs

Der Assembler im SEKA

Der Assembler wird über zwei Mechanismen gesteuert: Zum einen können Befehle (sogenannte Pseudo-Opcodes) in den Programmtext geschrieben werden, zum anderen sind einige Optionen nach dem Befehl A (für Assemble) möglich. Beginnen wir mit den Optionen des Assemblers:

v Ausgabe des Listings auf den Bildschirm.

p Ausgabe des Listings auf den Drucker.

h Ausgabe erfolgt seitenweise, weiter auf Tastendruck.

o optimiert relative Sprünge (Branches).

l erzeugt linkfähigen Code.

Die folgenden Opcodes steuern die Arbeit des SEKA und werden direkt in den Programmtext eingefügt:

dc fügt Daten ein, wird keine Länge angegeben (".b", ".w", ".l"), verwendet der SEKA automatisch ".b". Texte können in Hochkomma oder Anführungszeichen eingegeben werden.

```
dc.w 10,10,20
dc.b "dos.library",0
```

blk reserviert die angegebene Anzahl Bytes, Worte oder Langworte und füllt den Bereich optional mit einem Wert.

```
blk.l 100               ; 100 Langworte mit beliebigem Inhalt
blk.b 10,0              ; 10 Bytes mit dem Inhalt 0
```

org bestimmt die Adresse, für die das Programm assembliert werden soll. Dadurch wird absoluter, also nicht mehr verschiebbarer Code erzeugt, der nur noch an die Adresse geladen und dort gestartet werden kann.

code assembliert ins Code-Segment, schaltet automatisch relative Assemblierung (verschiebbar) ein.

data assembliert ins Datensegment.

even fügt gegebenenfalls ein Füllbyte ein, damit die Adresse in jedem Fall gerade ist.

odd macht Adresse im Gegensatz zu even ungerade.

end beendet die Assemblierung.

equ Zuweisung von Werten an ein Label

= dieselbe Wirkung wie equ.

list schaltet Listing ein.

nlist schaltet Listing aus.

page Beginn einer neuen Seite des Listings (Seitenvorschub).

if ist der folgende Parameter 0, wird bis zum endif nicht mehr assembliert, dient zur bedingten Assemblierung

else Gegenstück zu if, assembliert, falls hinter if der Parameter 0 war, ansonsten nicht.

endif Ende der bedingten Assemblierung.

```
; Beispiel für bedingte Assemblierung

Deutschland = 1         ; deutsche Meldungen

Fehler:
if Deutschland
dc.b "Falsche Diskette!",0
```

```
else
dc.b "wrong diskette",0
endif
```

macro
Beginn einer Makro-Definition; vorher wird ein Label angegeben.

endm Ende einer Makro-Definition.

?n wird durch Makroargument n ersetzt, möglich sind 1-9.

?0 erzeugt für jeden Makro-Aufruf eine neue dreistellige Ziffernfolge, mit der lokale Labels erzeugt werden können.

ifb ist wahr, wenn das dahinter angegebene Makro-Argument leer, also nicht angegeben ist.

```
LIST E                      ; Listing einschalten, Makroaufrufe
                            ; erweitern und anzeigen

; ***** beim Assemblieren als Option v und h angeben

CALLEXEC: MACRO             ; Makro: Exec-Funktion aufrufen
MOVE.L A6,-(A7)             ; Inhalt von A6 auf dem Stack retten
MOVE.L 4,A6                 ; Execbase nach A6
JSR ?1(A6)                  ; Mit 1. Parameter als Offset
MOVE.L (A7)+,A6             ; A6 zurückholen
ENDM                        ; Makro-Ende

Openlibrary = -$0228        ; Exec: Bibliothek öffnen

Start:
MOVE.L #Dosname,A1          ; "dos.library"
MOVE.L #0,D0                ; beliebige Versionsnummer
CALLEXEC OpenLibrary        ; Makro aufrufen
TST.L D0                    ; hat alles geklappt ?
ILLEGAL
Dosname: dc.b "dos.library",0
```

globl definiert globale Labels, die innerhalb anderer Module verwendet werden können. Nur erlaubt, wenn mit Option L assembliert wird.

pwid Breite der Listingausgabe für Drucker, Standard ist 80.

plen Anzahl Zeilen bei der Druckausgabe, Standard ist 66.

pinit Steuersequenz für den Drucker vor der Ausgabe des Listings.

illegal erzeugt ungültigen Befehl $4AFC, mit dem der SEKA am Ende eines Programms wieder die Kontrolle erhält. Das illegal unbedingt durch ein RTS ersetzen, wenn das Programm vom CLI aus gestartet werden soll.

align sorgt dafür, daß die aktuelle Adresse durch eine angegebene Zweierpotenz teilbar ist.

```
align 4                 ; durch 4 teilbar
Langwort: dc.l 0
```

Der Debugger des SEKA

Mit dem Debugger können Programme im Einzelschritt untersucht werden. Folgende Kommandos sind möglich:

x Anzeige der aktuellen Register des 68000, wird dahinter ein Registername angegeben, kann dessen Inhalt anschließend neu festgelegt werden.

g startet das Programm an der aktuellen Adresse; wahlweise kann eine Adresse oder ein Label dahinter angegeben werden. Anschließend können noch Abbruchpunkte angegeben werden. Am besten setzt man vorher an die gewünschten Stellen Labels und gibt deren Namen dann als Breakpoints an.

j wie g, jedoch wird das Programm als Unterprogramm gestartet und kann mit RTS enden.

q Ausgabe von Speicherstellen ab der aktuellen oder dahinter angegebenen Adresse.

n gibt Speicherbereiche als 68000-Befehle aus (disassembliert).

a direktes Assemblieren an der angegebenen Adresse.

m ab der angegebenen Adresse können Speicherinhalte direkt geändert (modifiziert) werden. Abbruch der Änderung mit <Esc>.

s Abarbeitung eines einzelnen Befehls an der aktuellen Adresse (des Programm-Counters). Wahlweise kann dahinter die Anzahl der Einzelschritte angegeben werden.

f füllt Speicherbereiche, die Angaben Begin, End, Data werden anschließend erfragt.

c wie f, es wird aber ein Speicherbereich kopiert.

? Ausgabe eines beliebigen Ausdrucks, Labels oder ähnliches mit seinem Wert.

! Beendet die Arbeit mit dem SEKA nach einer Sicherheitsabfrage.

10.2.2 Der Profimat Assembler

Der zweite interessante Assembler ist der Profimat von DATA BECKER.

Vorteile des PROFIMAT

Der Profimat ist ebenfalls ein schneller Assembler, der direkt im Speicher des Amiga assemblieren kann. Wie beim SEKA erhalten Sie eine komplette Entwicklungsumgebung mit:

- Assembler
- Debugger
- Editor
- Reassembler

Der Profimat nutzt die Oberfläche des Amiga vollständig aus, arbeitet also mit Fenstern, Pull-Down-Menüs und unterstützt die Maus. Außerdem hat er ein spezielles Hilfefenster, in dem Sie sich die 68000-Befehle mit den möglichen Adressierungsarten oder die Unterprogramme der Betriebssystem-Bibliotheken mit den Übergabeparametern anschauen können.

Der mit dem Profimat mitgelieferte Reassembler stellt übrigens im Bereich der vorhandenen Amiga-Assembler etwas besonderes dar. Mit ihm können Sie (mit gewissen Einschränkungen) fremde Programme in Programmtext zurückverwandeln. Dadurch können Sie diese noch einfacher anschauen und gegebenenfalls verändern.

Grundsätzliche Arbeit mit dem PROFIMAT

Den Profimat können Sie von der Workbench aus durch Anklicken starten oder vom CLI aus mit seinem Namen aufrufen. Nach dem Start öffnet er vier verschiedene Fenster, die sich wie auf dem Amiga üblich beliebig positionieren und in der Größe verändern lassen.

Im Editor-Fenster geben Sie das Programm ein. Durch Anklicken des Assembler-Fensters aktivieren Sie den Assembler und können nach Betätigung der rechten Maustaste aus den Pull-Down-Menüs die gewünschten Befehle wählen. Um ein Programm im Debugger zu testen, müssen Sie es nach dem Assemblieren speichern und dann in das Debugger-Fenster laden. Um im Debugger-Fenster alle Variablen und Labels angezeigt zu bekommen, müssen Sie im Menü "Debugger" den Befehl "Variablen berechnen" wählen und anschließend im Befehl "Parameter" "Anzeigen" die Einstellung "symbolisch".

Laden und Speichern von Programmen

Bestehende Programmtexte werden mit dem Menü "Datei" "Öffnen" geladen. Anschließend erscheint ein Dateiauswahlfenster, in dem die Datei ausgewählt werden kann. Ist schon ein Programmtext im aktuellen Editor-Fenster vorhanden, öffnet der Profimat einfach ein neues Fenster und lädt den Programmtext dort hinein. Es können also gleichzeitig mehrere Programme bearbeitet werden.

Um ein Programm aus dem Editor-Fenster zu entfernen und ein neues Programm zu laden, benutzen Sie das Menü "Datei" "Neu" oder klicken einmal in das Schließsymbol des Editor-Fensters. Anschließend können Sie das Programm mit "Datei" und "Öffnen" laden.

Um ein Programm zum erstenmal oder unter einem neuen Namen zu speichern, verwenden Sie den Befehl "Datei" "Abspeichern als". Anschließend erscheint wieder ein Dateiauswahlfenster, in dem Verzeichnis und Dateiname eingegeben werden

können. Um ein Programm unter einem bestehenden Namen zu speichern, verwenden Sie den Befehl "Datei" "Abspeichern".

Assemblieren und Starten

Haben Sie das Programm im Editor eingegeben und gespeichert, können Sie es assemblieren, in dem Sie das Assembler-Fenster aktivieren und den Befehl "Assembler" "Assemblieren" verwenden. Anschließend erscheint ein Fenster, in dem festgelegt werden kann, in welchem Speicher die einzelnen Teile des Programms abgelegt werden sollen. Da dies für unsere Programme keine Rolle spielt, wählen Sie "OK". Enthält das Programm keine Fehler, können Sie es anschließend mit "Datei" "Abspeichern" abspeichern. Im Gegensatz zum Programmtext, dem wir ja immer die Erweiterung ".S" geben, wird es dabei ohne Erweiterung gespeichert.

Sollte ein Fehler im Programm vorhanden sein, erscheint ein Fenster mit der fehlerhaften Zeile und drei Möglichkeiten:

Mit "Abbruch" brechen Sie den Vorgang ab, mit "Nochmal versuchen" wird eine Änderung der fehlerhaften Zeile für einen erneuten Versuch verwendet, aber nicht im Editor dauerhaft gespeichert, dies erfolgt mit der dritten Möglichkeit "Abspeichern und nochmal versuchen".

Nach dem erfolgreichen Assemblieren und Speichern können Sie das Programm direkt vom CLI aus mit seinem Namen aufrufen, oder in den Debugger laden, um es unter seiner Kontrolle zu starten. Für den zweiten Fall aktivieren Sie das Debugger-Fenster und wählen dort den Befehl "Debugger" "Laden".

Um im Debugger-Fenster alle Variablen und Labels angezeigt zu bekommen, müssen Sie im Menü "Debugger" den Befehl "Variablen berechnen" wählen und anschließend im Befehl "Parameter" "Anzeigen" die Einstellung "symbolisch" wählen. Im Menü "Steuerung" des Debuggers finden Sie drei Menüpunkte, mit denen das Programm gestartet werden kann:

Start Das Programm wird gestartet und läuft mit hoher Geschwindigkeit bis zum Programmende oder einem gewählten Abbruchpunkt (Breakpoint).

Start abbrechbar Das Programm wird Schritt für Schritt automatisch abgearbeitet und kann jederzeit durch den Menüpunkt "Steuerung" "Abbrechen" gestoppt werden.

Nächster Befehl Es wird nur ein Befehl des Programms abgearbeitet.

Um einen Abbruchpunkt zu setzen oder zu löschen, klicken Sie die gewünschte Adresse im rechten Fenster an und wählen im Menü "Steuerung" "Breakpoint" entweder "setzen" oder "löschen". Wird das Programm anschließend mit "Steuerung" "Start" gestartet, unterbricht der Debugger die weitere Abarbeitung, sobald diese Adresse erreicht wird.

Die wichtigen Unterschiede zum SEKA

Wenn Sie unsere Beispielprogramme mit dem Profimat eingeben und assemblieren wollen, gibt es einige Unterschiede zu beachten:

- Jede Befehlszeile muß mit einem Leerzeichen beginnen. Die einzigen Zeilen, die mit dem ersten Zeichen beginnen dürfen und müssen (!) sind Zeilen mit einem Label. Auch Kommentarzeilen dürfen mit einem Semikolon in der ersten Spalte beginnen. Ein für den SEKA geschriebenes Programm müßte also folgendermaßen umgeschrieben werden:

```
; Programmelement für den SEKA
; **** Öffnen der Dos.library ****
OpenDos:
MOVE.L Execbase,A6        ; Basisadresse nach A6
MOVE.L #Dosname,A1        ; Libraryname nach A1
MOVE.L #0,D0              ; 0 = beliebige Versionsnummer
JSR OpenLibrary(A6)
RTS
```

```
; Programmelement für den Profimat
; **** Öffnen der Dos.library ****
OpenDos:
 MOVE.L Execbase,A6        ; Basisadresse nach A6
 MOVE.L #Dosname,A1        ; Libraryname nach A1
 MOVE.L #0,D0              ; 0 = beliebige Versionsnummer
 JSR OpenLibrary(A6)
 RTS
```

- Das Hochkomma (') ist im Profimat nicht erlaubt, verwendet Sie statt dessen die Anführungszeichen (").

- Der Befehl EVEN ist nicht erlaubt, verwenden Sie statt dessen für die Ablage von Daten an einer geraden Adresse den Befehl ALIGN.

- In den Programmen zum Öffnen eines Intuition-Fensters verwenden wir das Zeichen "!" zur Oder-Verknüpfung von Zahlen. Dies ist im Profimat nicht erlaubt - verwenden Sie statt dessen das Zeichen "^" rechts neben dem "Ä".

- Sie müssen in die letzte Zeile des Programms den Befehl END schreiben. Damit er nicht als Label, sondern als Befehl erkannt wird, muß die Zeile mit einem Leerzeichen beginnen.

- Der letzte Befehl des Programms sollte kein ILLEGAL, sondern der Befehl RTS sein. Wird im Debugger dieses letzte RTS des Programms erreicht, betrachtet dieser das Programm als beendet. Um es erneut starten zu können, müssen Sie es wieder in den Debugger laden.

10.2.3 Andere Assembler

Es gibt noch einen weitereren Assembler für den Amiga, von denen wir zwei kurz erwählen wollen, weil sie sich erheblich vom Profimat oder SEKA unterscheiden. Beide haben keine integrierte Entwicklungsumgebung, sondern das Programm wird mit einem beliebigen Editor eingegeben, durch den Aufruf des Assemblers vom CLI aus assembliert und kann anschließend gestartet werden.

Der ASSEM

Der ASSEM von Metacomco wurde am Anfang mit dem Entwicklungspaket zum Amiga ausgeliefert. Nach dem Erstellen des Programms mit einem Editor starten Sie das Assemblieren mit folgender Befehlszeile im CLI:

```
ASSEM Quelldatei -o Zieldatei
```

Anschließend erhalten Sie eine Zwischendatei, die mit einem Linker (beispielsweise dem ALINK) in ein ausführbares Programm umgewandelt werden muß. Die Befehlszeile dafür lautet:

```
ALINK Zieldatei TO Programmname
```

Die wichtigsten Unterschiede zum SEKA

- Verwenden Sie statt des Gleichheitszeichens zur Wertzuweisung die Zeichenfolge EQU.
- Verwenden Sie statt EVEN die Befehlsfolge CNOP 0,2.
- Wenn Sie Zeichenketten eingeben wollen, verwenden Sie statt der Anführungszeichen das Zeichen "'" (Eingabe mit ALT + "ä").

10.3 Pannenhilfe

Es gibt Programmiersprachen, bei denen Fehler bei der Programmierung höchstens zum "Syntax Error" führen. Dagegen ist Maschinensprache oder Assembler eher wie ein Drahtseilakt ohne Netz und doppelten Boden. Nicht umsonst haben die Entwickler des Amiga-Betriebssystems die berühmten endgültigen Fehlermeldungen "Guru Meditation" genannt, denn das anschließende geduldige Warten darauf, daß der Amiga wieder betriebsbereit ist und man den Fehler suchen kann, verlangt viel Geduld.

Damit Sie bei möglichen Pannen möglichst schnell eine Lösung finden, haben wir die wichtigsten Pannen in diesem Kapitel zusammengefaßt.

Programm wird nicht abgearbeitet

Problem: Sie starten das Programm, aber offensichtlich wird es nicht abgearbeitet.

Lösung: Wahrscheinlich haben Sie das Programm nicht assembliert. Holen Sie dies mit dem Befehl A nach. Besonders leicht passiert dieser Fehler, wenn man ein Programm nach dem ersten Start erneut starten will.

Programm stürzt nur im CLI ab

Problem: Sie haben ein Programm geschrieben, und im SEKA funktioniert es tadellos. Sobald Sie es aber mit dem Befehl WO speichern und vom CLI aus aufrufen, erscheint der Requester "Task held, finish all...".

Lösung: Wahrscheinlich haben Sie vergessen, den Befehl ILLEGAL am Programmende durch ein RTS zu ersetzen. Der Befehl ILLEGAL darf nur im SEKA verwendet werden.

Wenn Ihnen dieser Fehler häufiger passiert, verwenden Sie ILLEGAL auch im SEKA nicht mehr. Benutzen Sie statt dessen den Befehl RTS, und starten Sie Ihre Programme nicht mehr mit G (für GO), sondern mit J (für JSR).

Änderung oder Abfrage eines Bits tuts nicht

Problem: Sie überprüfen ein Bit einer Speicherstelle oder ändern es, aber das gewünschte Ergebnis tritt nicht ein.

Lösung: Wenn Sie die richtige Speicherstelle genommen haben, kann der Fehler beispielsweise an der Numerierung der Bits liegen. Das 1. Bit wird mit der

Nummer 0 angegeben, das zweite mit 1 und so fort. Um also das zweite Bit der Speicherstelle $BFE001 zu ändern, müssen Sie "BCHG #1,$BFE001" schreiben.

Fehlermeldung: Illegal Operator

Problem: Der SEKA meldet beim Assemblieren die Fehlermeldung "Illegal Operator".

Lösung: Für diesen Fehler gibt es eine Reihe von Möglichkeiten: Grundsätzlich haben Sie einen ungültigen Befehl verwendet. Überprüfen Sie noch einmal genau die Schreibweise der Befehle in der Befehlszeile. Dieser Fehler tritt auch auf, wenn Sie den Doppelpunkt hinter einem Label vergessen haben.

Fehlermeldung: Illegal Adressing Mode

Problem: Sie erhalten beim Assemblieren die Meldung: "Illegal Adressing Mode".

Lösung: Sie haben versucht, mit einem Adreßregister zu rechnen, oder mit einem Datenregister eine Speicherstelle zu adressieren.

Programm arbeitet nicht korrekt: Zahlenproblem

Problem: Sie wollen mit Zahlen rechnen, stellen aber fest, daß ganz andere Zahlen, als die von Ihnen gewünschten, verwendet werden.

Lösung: Überlegen Sie noch einmal genau, in welchen Zahlensystem Sie die Zahlen eingegeben haben. Wollen Sie Dezimalzahlen eingeben, so darf vor der Zahl nichts stehen, bei Hexadezimalzahlen muß vor die Zahl das $-Zeichen und bei Dualzahlen das %-Zeichen.

Programm funktioniert nicht oder produziert einen "Adress Error"

Problem: Nach dem Start funktioniert Ihr Programm nicht so, wie Sie sich das gedacht haben, oder Sie erhalten die Fehlermeldung: Adress Error.

Lösung1: Sie wollen mit einem Befehl eine Adresse in ein Register schreiben, haben aber das Zeichen "#" vor dem Label vergessen.

Lösung2: Sie haben bei einer Wort- oder Langwortoperation versucht, den Inhalt einer Speicherstelle an einer ungeraden Adresse zu bearbeiten. Ändern Sie die Adressierung in Byte-Länge, oder benutzen Sie den Befehl EVEN, um eine gerade Adresse zu erhalten.

Programm endet nicht

Problem: Sie warten und warten, und das Programm endet nicht.

Lösung: (Hoffentlich hatten Sie es abgespeichert!) Wahrscheinlich enthält Ihr Programm eine Endlosschleife. Meist passiert dies dadurch, daß keine oder eine fehlerhafte Abbruchbedingung in der Schleife vorhanden ist.

Zahlenproblem bei Initialisierung

Problem: Sie haben nicht die erwarteten Zahlenwerte in den Registern, und das Programm arbeitet nicht wie vorgesehen.

Lösung: Entweder haben Sie eine Variable nicht oder fehlerhaft initialisiert. Wenn Sie ein Register beispielsweise mit MOVE.B #0,D0 "auf 0 setzen", kann in den oberen Bits immer noch eine beliebige Zahl stehen, denn diese wurde durch die Angabe .B nicht betroffen. Benutzen Sie sicherheitshalber zum Initialisieren die Adressierungslänge Langwort.

Hauptprogramm wird mehrfach abgearbeitet, Programmabsturz

Problem: Merkwürdigerweise wird ein Teil des Hauptprogramms mehrfach abgearbeitet.

Lösung: Möglicherweise haben Sie ein Unterprogramm nicht ordnungsgemäß mit RTS verlassen, sondern sind mit JMP ins Hauptprogramm zurückgesprungen. Dadurch hat der Prozessor ein RTS zu viel auf dem Stack.

Nach einem Unterprogrammaufruf tritt ein Fehler auf

Problem: Nachdem Ihr Programm ein Unterprogramm aufgerufen hat, arbeitet es nicht mehr korrekt, sondern beispielsweise mit falschen Zahlen weiter.

Lösung: Möglicherweise haben Sie nicht genau überlegt, welche Register für welche Zwecke dienen und das Unterprogramm ändert Registerinhalte, die im Hauptprogramm noch benötigt werden.

Library kann nicht geöffnet werden

Problem: Sie versuchen, mit der Exec-Funktion OpenLibrary eine andere Bibliothek zu öffnen, erhalten aber als Rückgabewert immer eine 0 in D0.

Lösung: Entweder haben Sie den Namen der Library falsch geschrieben (komplett in kleinen Buchstaben), oder Sie haben eine falsche Versionsnummer angegeben. Wenn Sie auf die Version keinen Wert legen, löschen Sie das Register D0 bitte mit MOVE.L #0,D0 (wichtig ist auch das .L).

Beim Aufruf einer Bibliotheksroutine stürzt Ihr Programm ab

Problem: Sie haben die gewünschte Library geöffnet, aber nach dem Aufruf der Unterroutine stürzt das Pro-

gramm ab. Alle Übergabewerte für die Unterroutine sind in Ordnung, und bis vor der letzten Änderung hat das Programm tadellos funktioniert.

Lösung: Wahrscheinlich haben Sie nicht darauf geachtet, vor dem Aufruf der Library-Funktion die Basisadresse nach A6 zu holen. Nun haben Sie einen anderen Bibliotheksaufruf in Ihr Programm eingefügt, so daß in A6 eine falsche Basisadresse steht. Am besten setzen Sie die Basisadresse explizit vor jedem Aufruf einer Betriebssystem-Funktion.

Lösung: Sie haben versehentlich bei der Übergabe von Parametern an das Betriebsystem eine falsche Operandenlänge verwendet. Fast alle Betriebssytemroutinen erwarten Langworte.

Absturz beim Schließen einer Library oder eines Fensters

Problem: Sie erhalten innerhalb des Programms einen Fehler, der zum Programmende führen soll, doch beim Schließen der geöffneten Objekte stürzt das Programm ab.

Lösung: Sie versuchen in dem Programmteil etwas zu schließen, das vorher gar nicht geöffnet wurde. Sie springen also wahrscheinlich zu einem falschen Label innerhalb der Routine "Schließen". Man kann sich vor solchen Problemen schützen, in dem man vor dem Schließen prüft, ob man ein gültiges Handle (ungleich null) erhalten hat. Beispiel:

```
Schließen:
MOVE.L IntuitionBase,A6       ; Basisadresse Intuition.library
MOVE.L WinHandle,A0           ; Handle des (geöffneten?)
                              ; Fensters
TST.L WinHandle               ; Ist das Fenster wirklich offen?
BEQ Ende                      ; Nein, also auch nicht schließen
JSR CloseWindow(A6)           ; Es ist offen, Fenster schließen
Ende:
```

Probleme nach Programmänderung

Problem: Nach einer Programmänderung funktioniert das Programm insgesamt nicht mehr.

Lösung: Sie haben für bestimmte Zahlenwerte oder Adressen keine Konstanten (Labels) benutzt und dadurch vergessen, einige Werte im Programm zu ändern. Diese Programmteile arbeiten also noch mit den alten Werten.

10.4 Lexikon

Das folgende Lexikon soll Ihnen eine Hilfe sein, wenn Ihnen ein in diesem Buch erklärter Begriff nicht mehr ganz klar ist und Sie nicht die zugehörige ausführliche Erklärung lesen wollen.

Absolute Adressierung Bei der absoluten Adressierung wird der Inhalt einer Speicherstelle direkt angesprochen. Dabei wird beim absoluten Adressieren einer Quelle deren Inhalt geholt, bei der absoluten Adressierung eines Ziels in die Speicherstelle hineingeschrieben. Im Befehl wird also direkt eine Adresse oder ein Label angegeben.

Adreßbus Siehe -> Bus

Adreßregister Adreßregister dienen zum Adressieren von Operanden. Beim Ändern von Adreßregistern ist nur eine Wort- oder Langwortverarbeitung zulässig. Es gibt die Adreßregister A0 bis A7. Das Adreßregister A7 enthält den aktuellen Stackpointer, der auf die nächste, gültige Rücksprungadresse (RTS) zeigt.

Adreßzähler

Der Adreßzähler, in der Assemblersprache PC für Programm Counter, enthält immer die Adresse des nächsten Befehls. Sie können diese Adresse im SEKA mit XPC erfragen.

ALIGN

Siehe -> EVEN

Assembler

Ein Assembler ist ein Programm, das die von Ihnen eingegebenen Befehle in direkte Maschinensprachebefehle übersetzt, die der Prozessor versteht.

Assemblieren

Jedes Assembler-Programm muß, bevor es gestartet wird, assembliert, also in einen für den Prozessor verständlichen Code umgewandelt werden. Der Assembler rechnet dabei auch die Sprungadressen und Distanzen der Labels aus.

Basisadresse

Mit der Basisadresse ist die Adresse einer Bibliothek (Library) gemeint. Alle Unterroutinen dieser Bibliothek haben relativ zur Basisadresse immer denselben Abstand ->Offset.

Betriebssystem

Ein Betriebssystem ist ein Programm, das einen Computer überhaupt erst einmal in die Lage versetzt, zu arbeiten. Dieses befindet sich beim Amiga im ROM. Siehe also auch -> ROM.

Bibliothek

Eine Bibliothek, auch Library genannt, ist eine Sammlung von Unterroutinen, die zu ein und demselben Gebiet gehören. Bibliotheken haben jeweils feste Namen, an denen man ihren Zweck erkennen kann. Beispielsweise: DOS-Biblio-

thek, Grafik-Bibliothek usw. Bevor man eine Bibliothek benutzen darf, muß man Sie öffnen (Ausnahme: Exec.library).

Bit

Ein Bit ist die kleinste Informationseinheit in der Datenverarbeitung. Es gibt Aufschluß darüber, ob Strom fließt oder nicht, und kann die Werte 0 und 1 annehmen.

Breakpoint

Ein Breakpoint (Abbruchpunkt) ist eine Stelle im Programm, bei der die weitere Bearbeitung abgebrochen wird. Schreiben Sie ein Label an die Stelle im Programm, an der dieses unterbrochen werden soll, und geben Sie den Namen dieses Labels nach dem G im SEKA als Breakpoint an.

Bus

Zusammengehörende Leitungen, die elektronische Bauteile des Amiga miteinander verbinden, werden Bus genannt. Der Amiga hat zwei verschiedene Busse: einen Adreß- und einen Datenbus. Mit dem Adreßbus wird die zu bearbeitende Speicheradresse festgelegt. Mit dem Datenbus kann der Inhalt dieser Adresse gelesen oder verändert werden.

Byte

Eine Gruppe von 8 Bits, die gemeinsam bearbeitet werden. Ein Byte kann Werte zwischen 0 und 255 annehmen. Soll mit einen Befehl nur ein Byte bearbeitet werden, so wird dies durch ein .B gekennzeichnet.

CHIP-Memory

Das CHIP-Memory stellt die ersten 512 KByte des gesamten Speichers im Amiga dar. Nur diese können von den speziellen Custom-Chips (Blitter, Copper) bearbei-

tet werden. Daher müssen beispielsweise die Speicherbereiche, die die Bildschirminformation enthalten (Bitplanes) im CHIP-Memory liegen.

CPU

Der im Amiga befindliche 68000-Prozessor heißt CPU (Central Processing Unit = Zentrale Verarbeitungseinheit). Seine wichtigste Fähigkeit ist das Rechnen.

Dateimodus

Der Dateimodus spielt beim Öffnen von Dateien auf einer Diskette eine Rolle. Der Modus kann 1005 (Alt) oder 1006 (Neu) lauten, je nachdem, ob eine bestehende Datei (Alt) oder eine neue Datei (Neu) geöffnet werden soll. Wird eine bestehende Datei mit dem Modus Neu geöffnet, wird sie gelöscht und eine neue, leere Datei erzeugt - also Vorsicht!

Datenbus

Siehe -> Bus

Datenregister

Datenregister dienen zur Verwaltung von und besonders zum Rechnen mit Daten. In Datenregistern können sämtliche Operationen mit Daten durchgeführt werden. Es gibt die Datenregister D0 bis D7.

Dezimalsystem

Das Dezimalsystem ist ein Zahlensystem, in dem zehn Ziffern zur Verfügung stehen und das die Basis 10 besitzt. Mit jeder Stelle weiter nach links verzehnfacht sich der Wert einer Ziffer.

Dos.library

Eine Bibliothek, in der sich Unterroutinen für Ein- und Ausgabeoperationen (Bildschirm, Disketten usw.) befinden.

Dualsystem

Das Dualsystem besteht aus zwei Ziffern und zwar 0 und 1, die Basis ist 2. Wird eine Ziffer eine Stelle nach links geschoben, verdoppelt sich ihr Wert.

Einzelschritt

Unter Einzelschritt versteht man die Art, Programme so ablaufen zu lassen, daß immer nur ein Befehl abgearbeitet wird und das Programm danach stoppt. Es werden einem dann Auskünfte über die Inhalte der verschiedenen Register gegeben und der nächste abzuarbeitende Befehl angezeigt.

Endlosschleife

Eine Schleife, die nie verlassen wird. Meist passiert dies dadurch, daß keine oder eine fehlerhafte Abbruchbedingung in der Schleife vorhanden ist. Beliebt ist auch folgende Endlosschleife:

```
Schleife:
MOVE.L #10000,D0
SUB.L #1,D0
BNE Schleife
RTS
```

EVEN

Muß man auf Grund einer Wort- oder Langwortoperation auf eine gerade Speicherstelle zugreifen, so geschieht dies, in dem man in den Assembler die Anweisung EVEN vor diese Operation schreibt. In einigen Assemblern muß statt EVEN ALIGN verwendet werden.

Exec.library

Die Exec.library ist so etwas wie eine Grundbibliothek. Sie ist eine Bibliothek, die immer zur Verfügung steht. Die Adresse der Exec.library ist die einzige Adresse, die in jedem Amiga an der gleichen Stelle abgelegt ist. Sie ist auch die einzige Library, die nicht geöffnet

werden muß. In ihr befinden sich sämtliche Unterroutinen zur Nutzung der weiteren Bibliotheken und zur Verwaltung des Multitasking.

FAST-Memory

Der Rest des Speichers, der nicht von den speziellen Custom-Chips genutzt wird, also alles außer den ersten 512 KByte. Auf diesen Teil des Speichers kann der Prozessor ohne jede Arbeitsteilung und somit schneller zugreifen.

Flag

Die verschiedenen Bits des Statusregisters werden Flags genannt. Diese Flags werden bei bestimmten Ergebnissen von verschiedenen Operationen gesetzt und können später abgefragt werden, um beispielsweise bedingte Sprünge durchzuführen.

Gadget

Gadget ist die Bezeichnung für sämtliche Elemente eines Fensters, die angeklickt werden können und bei denen dann eine bestimmte Funktion ausgelöst wird. Das Schließsymbol eines Fensters ist beispielsweise ein Gadget.

Handle

Ein Handle ist eine Zahl, die bei bestimmten Operationen vom Betriebssystem zurückgeliefert wird und die fest zu der Operation gehört. Anhand dieser Zahl kann beispielsweise überprüft werden, ob die Operation durchgeführt wurde oder nicht. Des weiteren wird diese Zahl, also das Handle, benötigt, um weitere Aktionen auf Grund der vorangegangenen Operation auszuführen. Beim Öffnen einer Datei erhält man beispiels-

weise so ein Handle und muß dies zur weiteren Bearbeitung der Datei mit dem Betriebssystem immer angeben.

Hauptprogramm

Ein Hauptprogramm ist der Teil des Programms, in dem die eigentliche Struktur des Programms festgelegt wird. Vom Hauptprogramm aus werden Unterprogramme aufgerufen, in denen die einzelnen Operationen abgearbeitet werden.

Hexadezimalsystem

Das Hexadezimalsystem ist ein Zahlensystem, was auf der Basis 16 beruht. Im Hexadezimalsystem stehen die Ziffern 0 bis 9 und weiter A bis F zur Verfügung. Mit jeder Stelle, die eine Ziffer weiter links steht, versechzehnfacht sich ihr Wert. Hexadezimalzahlen werden durch das $-Zeichen kenntlich gemacht.

IDCMP-Flags

Die IDCMP-Flags (Intuition Direct Comunication Message Port = Intuitions direkter Briefkasten) spielen bei der Programmierung mit Intuition eine große Rolle. Bei dieser Art der Programmierung werden viele Vorgänge vom Betriebssystem (also Intuition) kontrolliert. Wollen Sie nun über bestimmte Vorgänge trotzdem informiert werden, so müssen Sie dieses durch die IDCMP-Flags bekanntgeben. Jedes Bit in einem speziellen Langwort in der Fenstertabelle steht für ein Ereignis, über das Sie informiert werden wollen.

ILLEGAL

Dies ist ein unzulässiger Maschinensprachebefehl, der zum Abbruch eines Programms führt, und die Kontrolle wird wieder dem Assembler übergeben. Au-

ßerhalb des Assemblers (beispielsweise beim Starten des Programms vom CLI aus) führt er zu einem "Task held.."-Requester.

Initialisierung

Durch eine Initialisierung werden Registern oder Speicherstellen zu Beginn des Programms bestimmte Werte zugewiesen. Man bestimmt damit also einen Anfangswert für bestimmte Operationen. Dies ist beispielsweise wichtig, wenn man einen neuen Wert mit einem älteren vergleichen will.

Intuition

Intuition ist der Teil des Amigas Betriebssystems, der die Bedienung so einfach und komfortabel macht (Fenster, Menüs usw.). Bei der Programmierung von Intuition übergibt man die Kontrolle über beispielsweise Grafik, Fensteraufbau usw. an Intuition. Wird eine Information über bestimmte Vorgänge gewünscht, muß dieses mit Hilfe der -> IDCMP-Flags angegeben und im Programm abgefragt werden.

Label

Ein Label ist eine Marke im Programm, durch die man bestimmten Stellen innerhalb des Programms einen Namen geben kann. Die Definierung des Labels muß immer mit einem Doppelpunkt abgeschlossen werden. Diese so gekennzeichneten Stellen können dann beispielsweise mit JMP angesprungen werden.

Langwort

Die größte Einheit, zusammengesetzt aus 32 Bits, die gemeinsam bearbeitet werden

kann. Soll mit einem Befehl ein Langwort bearbeitet werden, wird dies durch ein .L hinter dem Befehl gekennzeichnet.

LED Die LED ist die Kontrollampe am Amiga, die eingeschaltet ist, wenn der Amiga betriebsbereit ist. Die Information über die LED befindet sich im zweiten Bit der Speicherstelle $BFE001.

Library Siehe -> Bibliothek

Maschinensprache Maschinensprache ist die eigentliche Sprache, die ein Computer versteht. Diese besteht nur aus Zahlen, die eingegeben werden müssen. Da dieses sehr kompliziert und aufwendig ist, nimmt man einen -> Assembler zu Hilfe, der einfache Befehle kennt und diese in Maschinensprache umsetzt.

Message-Port Ein Message-Port ist eine Art Briefkasten, in dem Nachrichten abgelegt werden. Mit einem speziellen Unterprogramm aus der Exec.library WaitPort wartet man darauf, daß eine Nachricht vorhanden ist. Es gibt beispielsweise auch ein Unterprogramm, mit dem man Nachrichten verschicken kann. Dazu muß man natürlich zunächst einmal den Empfänger bzw. seinen Briefkasten herausfinden.

Offset Die Abstände zwischen Basisadresse und Unterroutinen werden Offsets der Unterroutinen genannt. Da die Adresse der Unterroutinen vor der Basisadresse liegen, ist klar, daß die Offsets immer einen negativen Wert haben. Die einzige

feste Speicherstelle im Amiga ist die Adresse der Exec.library (4). Alle anderen Adressen müssen durch das Öffnen der Libraries ermittelt werden.

PC — Siehe -> Adreßzähler

Programmzähler — Siehe -> Adreßzähler

Prozessor — Mit Prozessor wird der Teil des Computers bezeichnet, in dem die eigentlichen Rechenoperationen ablaufen.

Quelle — Als Quelle oder Quelloperand wird der Teil in einer Befehlszeile benannt, der direkt hinter dem Befehl steht.

RAM — Der Teil des Speichers, in dem Informationen geändert werden können, also der für uns interessante Teil des Speichers.

Register — Register sind Speicherstellen im Computer, die zur Verarbeitung von Daten dienen. Siehe auch -> Adreßregister, Datenregister und Statusregister. Im Gegensatz zu den übrigen Speicherstellen befinden sich Register direkt im Prozessor und können schneller bearbeitet werden.

ROM — ROM ist der nicht änderbare Teil des Speichers, in dem sich das Grundprogramm des AMIGAS befindet. Das Grundprogramm ist das Programm, das nach dem Einschalten abgearbeitet wird und das das Einlesen weiterer Informationen von Diskette oder Festplatte ermöglicht.

Rückgabewert

Ein Rückgabewert dient zur Überprüfung, ob Operationen ausgeführt wurden oder nicht.

Schleifen

Ein Schleife ist ein Teil eines Programms, das so oft abgearbeitet wird, bis eine bestimmte Bedingung erfüllt ist.

Speicher

Allgemeine Bezeichnung für den Teil des Rechners, in dem Daten abgelegt werden können. Siehe auch -> RAM, ROM und WOM.

Sprungbefehl

Ein Sprungbefehl ist ein Befehl, bei dem der Prozessor nicht mit dem nächsten Befehl fortfährt, sondern an eine bestimmte Adresse springt. Dort wird dann mit der Abarbeitung des Porgramms fortgefahren.

Stack

Der Stack ist ein spezieller Speicherbereich, in dem der Prozessor wichtige Daten wie beispielsweise Rücksprungadressen aus Unterprogrammen ablegt.

Statusregister

Das Statusregister ist das Register, in dem die -> Flags verwaltet werden.

Taktfrequenz

Die Taktfrequenz ist ein Maß für die Arbeitsgeschwindigkeit des Rechners. Alle elektronischen Bauteile arbeiten sozusagen im gleichen Rythmus (dem Takt), der von einem Quarz bestimmt wird. Beim Amiga beträgt die Taktfrequenz ungefähr 7,1 MHZ.

Task

Mehrere Programme, die gleichzeitig ablaufen, werden Tasks genannt.

Unterprogramm

Ein Unterprogramm ist ein Teil des gesamten Programms, der vom -> Hauptprogramm aus angesprungen und von dem aus auch wieder ins Hauptprogramm zurückgesprungen wird. In den Unterprogrammen werden die eigentlichen Operationen eines Programms durchgeführt.

Versionsnummer

So wie das Amiga-Betriebssystem eine Versionsnummer hat (aktuell ist heute 1.3), so haben auch die einzelnen Bibliotheken eine Versionsnummer, die bei bestimmten Operationen angegeben werden muß. Kennen Sie diese Versionsnummer nicht, oder spielt diese keine Rolle, geben Sie hier eine 0 ein.

Warteschleife

Unter einer Warteschleife versteht man eine Schleife, die nur den Sinn hat, den Ablauf eines Programms zu verzögern, die also eine bestimmte Zeit wartet, bevor mit der eigentlichen Arbeit im Programm fortgefahren wird.

WOM

WOM ist die Abkürzung für "Write Once Memory" (zu deutsch: einmal beschreibbarer Speicher) im Amiga 1000. Es ist also Speicher, in den nur einmal etwas hineingeschrieben werden kann und der dann nicht mehr änderbar ist. Vor dem ersten Beschreiben ist er also wie -> RAM, nach dem ersten Beschreiben wie ->ROM.

Wort

Eine Einheit, zusammengesetzt aus 16 Bits, die gemeinsam bearbeitet werden

kann. Soll mit einem Befehl ein Wort bearbeitet werden, so wird dies durch ein .W hinter dem Befehl gekennzeichnet.

Zeroflag

Das Zeroflag ist ein bestimmtes Bit im Statusregister. Es wird gesetzt, wenn das Ergebnis einer Operation gleich null ist.

Ziel

Als Ziel oder Zieloperand wird der Teil in einer Befehlszeile benannt, der hinter der -> Quelle durch ein Komma getrennt wird. Hierhin wird meistens das Ergebnis einer Operation geschrieben.

Stichwortverzeichnis

! 149
.... 30f, 51, 176
$ 31, 50
% 50
& 176
* 104, 178
.B 31
.COD 119
.H 173
.L 31
.S 47, 107
.W 31
; 62
? 37, 50

Abbruch eines Programms 34
Absolute Adressierung 51
ADD 32
Adresse, gerade 89
Adressierung 50, 120
Adressierungsart 88
Adreßbus 25
Adreßregister 26, 52
Adreßzähler 36
AKTIVATE 150
Aktuelles CLI-Fenster 104
Amiga 500 20
Amiga 1000 20
Amiga 2000 20
Anzahl gelesener Zeichen 113
Anzeige des Speichers 103
Art des Speichers 19
ASL 126
Assembler 29
Assembler-Directive 84
Assemblieren 32
Aufruf einer Library-Funktion 88
Aufruf eines Unterprogramms 58
Ausrufezeichen 149

Basisadresse 83, 89
Bausteine 92
BCHG 74
Befehls-Modus 30
BEQ 73
Betriebssystem 20
BGT 125
Bibliothek 82, 85
Bibliothek öffnen 83
Bibliotheksname 88
Bildschirmausgabe 92
Bit 11
Bit prüfen 70
BLT 124
BNE 60
Breakpoint 33
BTST 70
Bus 24
Byte 12, 31

C 143, 173
Char 178
CHIP-Memory 21
CLI-Fenster 104
Close 96
CloseLibrary 90
CloseWindow 147
CMP 73
CON-Fenster schließen 93
CPU 26
CTRL + \ 93
Custom-Chip 21
CUSTOMSCREEN 151

Datei erstellen 120
Datei lesen 113
Datei löschen 138
Datei öffnen 109
Datei verschlüsseln 115
Dateien vergleichen 187

Dateierweiterung 47, 107
Dateimodus 95, 113, 120
Dateiname abschließen 119
Datenbus 25
Datenleitung 25
Datenregister 26, 52
DC 84
Delay 90, 152
DeleteFile 138
Dezimalsystem 12
Directive 84
Diskette entfernen 156
DISKINSERTED 157
DISKREMOVED 156
DOS-Fenster 91
DOS-Fenster öffnen 93
Dos.library 170
Draw 167
Dualsystem 12

Editor 30
Eingabe einer Zahl 122
Eingaben über Tastatur 100, 122
Einzelschritt 38
Element einer Struktur 176
Ende eines Programms 114
Endlosschleife 72
Endung 107
EOR 120
Ereignis 149, 152
EVEN 89
Exec.library 170

Farbe 148
FAST-Memory 21
Fehler finden 42
Fensterdefinition 147
Fensterhandle 147
Fenstertabelle 155
FindTask 182
Flag 60

Gadget 150
Gerade Adresse 89
Gerät 92
Geschwindigkeit eines Prozessors 24
Graphics.library 162
Großbuchstabe 124, 151
Grundprogramm 19

Handle 97, 99, 103
Hardware 170
Hauptprogramm 57
Hexadezimal 31
Hexadezimalsystem 13
Hintergrund 149

IDCMP-Flags 149
ILLEGAL 34, 106
Include-Dateien 173
Indirekte Adressierung 52, 120
Initialisierung 77
INT 178
Intuition-Fenster 91
Intuition-Window 144
Intuition.library 143
Invertiert 161

JMP 73
JSR 58

Kleinbuchstabe 124
Kommandozeile 171, 187
Kommentar 62
Konstante 54, 84
Koordinate 148

Label 37, 53, 58
Label anspringen 73
Laden eines Programms 47
Langwort 12, 31, 53, 89
LED 45, 170
Lesen von Zeichen 113
Library 82
Linefeed 100, 104
Linie zeichnen 167
Long 178
Löschen einer Datei 138
Löschen einer Zeile 46

Löschen eines Programms 47

Mausbewegungen 72, 170
Maustaste 69
Maximale Breite 151
MC68000 23
Message-Port 152, 155, 172
Minimale Breite 151
Modus 95, 112
MOVE 31, 167
Multiplizieren 126
Multitasking 171, 181

Nachricht 152
Name der Bibliothek 85, 88
Neue Zeile 100
NewWindow 148
NOP 40

Oder-Verknüpfung 149
Öffnen einer Bibliothek 83
Öffnen einer Datei 109
Offset 82, 89
Open 95
OpenLibrary 84
OpenWindow 147

Parameter 171, 187
PC 27, 36
Pixel 166
PrintText 157
Priorität 181
Programm abbrechen 34
Programm laden 47
Programm löschen 47
Programm speichern 47, 106
Programm starten 33, 106
Programm strukturieren 57
Programme ohne SEKA 106
Programmende 114
Programmzähler 27
Prozessor 21, 23
Prüfen des Schließsymbols 151
Prüfen eines Bits 70
Punkt zeichnen 166

Quelle 50

RAM 19
Rastport 161, 172
Read 100
Rechenzeit 152
Register 26
ROM 19
RTS 58
Rückgabewert 77, 96, 176
Rückgabewert überprüfen 85
Rücksprung 28
Rücksprungadresse 28

Schließsymbol 150, 172
Schließsymbol prüfen 151
Schreibschutz 109
SEKA 29
Semikolon 62
SetAPen 166
SetTaskPri 182
SHORT 175, 178
SOURCE-Datei 47
Speicher 19
Speicher anzeigen 103
Speichern eines Programms 47, 106
Sprung 73
Sprungadresse 57, 58
Sprungbefehl 65
Stack 28
Stackregister 28
Starten eines Programms 33, 106
Statusregister 27, 61, 70
Struktur 143, 174
Struktur eines Programms 57
SUB 60

Tabelle 174
Taktfrequenz 24
Task 171, 181
Task finden 182
Task-Struktur 182
Tastatur 92, 183
Tastatureingabe 100, 122
Text ausgeben 97, 157

Textdefinition 161
Tipparbeit ersparen 91
Titel eines Fensters 150

Umwandlung von Zahlen 14, 122
Unmittelbare Adressierung 51
Unterprogramm 57
Unterprogramm aufrufen 58
Unterprogramm einfügen 92
Unterroutine 91

Variable 64, 76, 176
Vergleich 73
Vergleichen von Dateien 187
Verschieben 126
Verschlüsseln einer Datei 115
Versionsnummer 85
Verzögerungsroutine 90

WaitPort 152, 155
Warten 152
Warteschleife 56, 59
Window-Tabelle 152
WINDOWCLOSE 150, 155
WINDOWDEPTH 150
WINDOWDRAG 150
WINDOWSIZING 150
Windowtitel 150
WO 106
WOM 20
Workbenchscreen 150f
Wort 12, 31
Write 97
WritePixel 166

Zahlen lesen 122
Zahlen umwandeln 14, 50
Zahlensystem 12, 50
Zähler 65
Zeichen, Anzahl gelesene 113
Zeichen lesen 113
Zeichenfarbe 160
Zeichenfläche 161
Zeichenmodus 161
Zeichenstift 167
Zeichnen einer Linie 167
Zeichnen eines Punktes 166
Zeiger 178
Zeilen löschen 46
Zeilenvorschub 100
Zeroflag 60, 65, 70
Ziel 50
Ziffer 124

Kompetentes Detailwissen zum Amiga 2000.

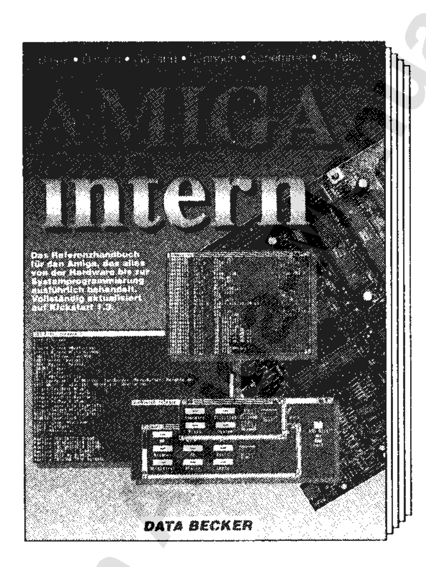
Von den glorreichen Drei (Rügheimer/Spanik/Amiga) wurden die Amiga-Anwender schon immer verwöhnt: Ihre Fachbücher strotzen vor fundiertem Know-how und sind dennoch dank einer lockeren Schreibe überaus unterhaltsam. Warum sollte es also beim großen Amiga-2000-Buch anders sein? Mit viel Liebe zum Detail und der gewohnten leichtverständlichen Sprache beschreiben Sie alles, was den Amiga 2000 so interessant macht: die möglichen Speichererweiterungen, Einbau und Einrichtung einer PC-/Amiga-Harddisk, Arbeiten mit einer PC-/AT-Karte, Kickstart im RAM, jede Menge zur Janus.Library 2.0, Software-Installationstips, der Umgang mit dem AmigaDOS und und und. Selbstverständlich auch mit einer detaillierten Einführung zum Amiga 2000 und einigen wichtigen Software-Tips. Das große Amiga-2000-Buch – ein großartiges Buch zu einem großartigen Rechner.



Rügheimer/Spanik
Das große Amiga-2000-Buch
Hardcover, 782 Seiten, DM 59,–
ISBN 3-89011-199-8

In einem Band: alle Details zu Ihrem Amiga.

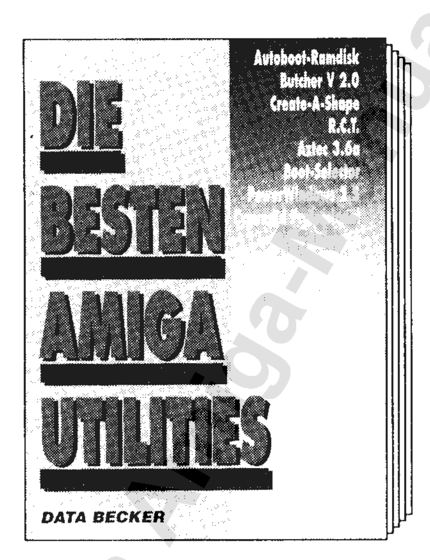
Amiga Intern – von der ersten bis zur letzten Seite bietet Ihnen dieses einzigartige Nachschlagewerk alle harten Facts zu Ihrem Rechner. Angefangen bei einer detaillierten Beschreibung des 68000-Prozessors, der CIA, der Custom-Chips und der Schnittstellen über die Hardware-Programmierung bis hin zu einer leichtverständlichen Dokumentation aller Library-Funktionen – zu allen bisher ausgelieferten Kickstart-Versionen. Aus dem Inhalt: die Strukturen von EXEC, I/O-Handhabung, Verwaltung der Resources, Erstellen eigener Devices, EXEC-Base, resetfeste Programme, DOS-Funktionen, interne DOS-Bibliothek, Aufbau einer Diskette, Autoboot mit der ROMboot.library, Programmierung eigener Handler, Ein- und Ausgabe über die verschiedenen Amiga-Devices, Standard-Austausch-Formate und IFF-Komprimierungsverfahren, alle Amiga-Libraries mit den dazugehörigen Strukturen, Basis- und Grundstrukturen, Preferences als Datenstruktur, Datenübermittlung von Workbench und CLI, Konventionen im Programmierstil...



Bleek/Dittrich/Gelfand/Jennrich/Schemmel/Schulz
Amiga Intern
Hardcover, 1095 Seiten, DM 98,-
ISBN 3-89011-398-2

Kleine Helfer, die sich sehen lassen können.

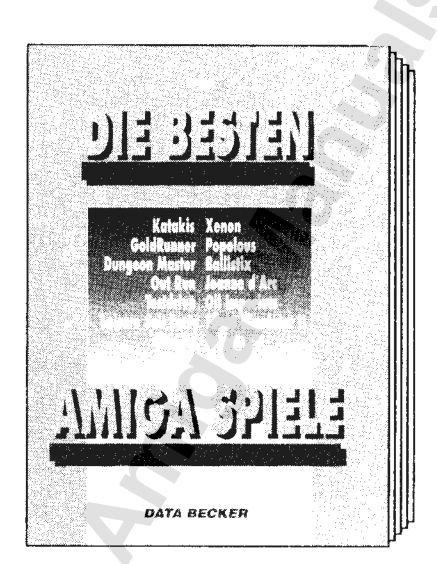
Utilities sind immer eine feine Sache – je nach Programm können sie die Arbeit am Rechner erheblich erleichtern, oder auch schon 'mal den einen oder anderen Fehler wieder gutmachen. Einziger Haken: „Dank" der meist unzureichenden Beschreibung ist man oft nicht in der Lage, den gesamten Leistungsumfang des jeweiligen kleinen Helfers zu überblicken. Daher dieses Buch: Die besten Amiga Utilities. Und tatsächlich bietet Ihnen dieser Band eine detaillierte Beschreibung der beliebtesten und stärksten Hilfsprogramme – von der Installation über die Bedienung bis hin zu nützlichen Tips. Hier die Utility-Hitliste: Diskmaster, Butcher V2, Discovery, der Editor CygnusEd Professional, Quarterback, Aztec C-Compiler, Power Windows, Create-A-Shape, Zenon und Zing!Keys. Eben alles, was in der Amiga-Utility-Szene Rang und Namen hat, wird in diesem Band besprochen. Umfassend, detailliert und mit vielen praktischen Anwendungshinweisen. Die besten Amiga-Utilities – das „Handbuch" zu Ihren Hilfsprogrammen.



Polk
Die besten Amiga Utilities
403 Seiten, DM 39,-
ISBN 3-89011-108-4

Spielen mit dem Amiga: das reinste Vergnügen.

Entspannung muß sein. Besonders, wenn man an einem Amiga arbeitet. Denn bei dieser Grafik, diesem Sound macht „Computerspielen" erst richtig Spaß. Kein Wunder also, daß es gerade in der Amiga-Welt so viele klangvolle Namen gibt, die ungetrübten Spiele-Spaß versprechen: Katakis, Populous, Leisure Suit Larry, SimCity, Falcon, Karate Kid II, California Games. In „Die besten Amiga-Spiele" werden sie und zahlreiche andere ausführlich besprochen – so werden hier die Hardware-Voraussetzungen und die Installation des Programms genauso beschrieben wie das Ziel und der Aufbau des jeweiligen Spiels. Zahlreiche praktische Tips und Lösungsvorschläge helfen zusätzlich, wenn Sie nicht mehr weiterkommen. Ganz gleich, ob Baller-, Strategie-, Abenteuer-, Geschicklichkeits-, Kampf- oder Sportspiele, ob Autorennen, Handelssimulationen oder Flugsimulatoren: Hier finden Sie die Informationen für das totale Spiele-Vergnügen. Die besten Amiga-Spiele – vielleicht die schönste Seite Ihres Computers.



Maelger
Die besten Amiga-Spiele
261 Seiten, DM 39,-
ISBN 3-89011-371-0

TEXTOMAT und DATAMAT : Das Power-Pack Amiga.



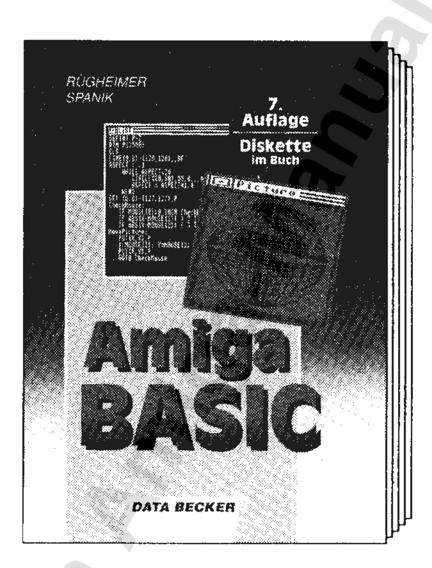
Für alle Amiga-Fans halten wir etwas ganz Besonderes parat: das Power-Pack Amiga! Es umfaßt unsere bewährten Programme TEXTOMAT und DATAMAT für den Amiga. TEXTOMAT ist die ideale Textverarbeitung für Anfänger und Textprofis. Das Programm bietet Ihnen z. B. die Möglichkeit, sämtliche Befehle über Menüleisten oder Kurzbefehle anzusteuern, Bildschirmausschnitte aus anderen Programmen einzulesen und zu verarbeiten, eine schnelle Direktformatierung, die Darstellung der Schriftattribute und Grafikeinbindung direkt auf dem Bildschirm, eine automatische Silbentrennung sowie fertige Druckanpassungen für alle gängigen Drucker. DATAMAT ist das bewährte Dateiprogramm, das Ihnen die Pflege und Verwaltung Ihrer Daten so einfach wie möglich macht. Zu den Leistungsmerkmalen von DATAMAT zählen eine Dateigröße von max. 2 Milliarden Zeichen, maximal 8 gleichzeitig zu öffnende Dateien, eine maximale Feldgröße von 32.000 Zeichen, maximal 80 Indexfelder mit wählbarer Genauigkeit, frei gestaltbare Bildschirmmasken, ein integrierter Druckmasken- und Listeneditor u.v.a.m.

TEXTOMAT Amiga
unverb. Preisempf. DM 99,-
ISBN 3-89011-580-2

DATAMAT Amiga
unverb. Preisempf. DM 99,-
ISBN 3-89011-581-0

Jede Menge Programme und Utilities.

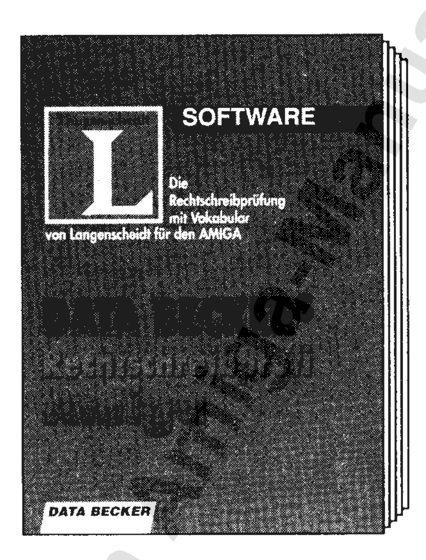
Fast 800 Seiten über AmigaBASIC - von Fans (das bekannte Duo Rügheimer/Spanik) für Fans. Im ersten Teil werden Sie Schritt für Schritt - und das vor allem auf verständliche Weise - in die Programmierung des Amiga eingeführt, im zweiten Teil finden Sie alle gelernten Befehle mit Syntax und Parameterangaben zum schnellen Nachschlagen. Dazu gibt es Programme und Utilities in Hülle und Fülle: ein Videotitel-Programm (OBJECT-Animation), ein Balken- und Tortengrafik-Programm, ein Malprogramm (mit Windows, Pulldowns, Mausbefehlen, Füllmustern und dem Einlesen sowie Abspeichern von IFF-Bildern), ein Statistikdaten-Programm, ein Sprach-Utility, ein Synthesizer-Programm u.v.a.m.



Rügheimer/Spanik
AmigaBASIC
Hardcover, 777 Seiten
inkl. Diskette, DM 59,-
ISBN: 3-89011-209-X

Alles korrekt? Fragen Sie den Rechtschreibprofi.

DATA BECKERs Rechtschreibprofi ist das schnelle Prüfprogramm für alle, die Wert auf einwandfreie Texte legen. Ganz gleich, welche Textverarbeitung Sie besitzen: Lassen Sie Ihre Berichte, Artikel, Protokolle, wissenschaftliche Texte und Diplom- oder Studienarbeiten bequem vom Computer „gegenlesen". Die komfortable Oberfläche erlaubt auch Computer-Neulingen die rasche Nutzung des Programms. Für die Richtigkeit des Prüfvorgangs garantiert ein bekannter Name: Langenscheidt-Mitarbeiter haben die rund 106.000 Wörter und Flexionen des Hauptwörterbuchs zusammengestellt. Da DATA BECKERs Rechtschreibprofi sich am Wortstamm orientiert, sind damit auch alle möglichen Kombinationen erfaßt: Insgesamt werden also 2,5 Millionen Wörter überprüft. Natürlich haben Sie außerdem die Möglichkeit, weitere Fachausdrücke und fremdsprachliche Begriffe im Benutzerlexikon abzulegen.



DATA BECKERs Rechtschreibprofi Amiga
ISBN 3-89011-585-3
DM 99,- (unverb. Preisempfehlung)

Maschinensprache für Einsteiger

Lassen Sie sich nicht von Assembler abschrecken, sondern finden Sie selbst heraus, was in Ihrem Rechner steckt. Denn Maschinensprache lernen heißt nicht, seitenweise Tabellen und Adressierungsarten auswendig zu lernen. Hier wird Ihnen eine leichtverständliche Einführung in die Maschinenprogrammierung des Amiga geboten, die keinerlei Vorkenntnisse voraussetzt, so daß Sie gleich von Anfang an ohne Fachchinesisch Spaß am Programmieren haben. Natürlich zeigt Ihnen dieses Buch auch, wie schnell man zum Amiga-Profi aufsteigen kann, denn ein entsprechendes Aufsteigerkapitel rundet das Werk ab. Und ganz nebenbei werden Sie auch noch mit dem gebräuchlichsten Assembler auf dem Amiga vertraut gemacht, dem SEKA-Assembler. Die verschiedenen Anhänge bieten Ihnen schließlich eine Pannenhilfe, denn nicht alles klappt gleich beim erstem Mal, ein umfangreiches Fachwortlexikon und natürlich eine komplette Befehlsreferenz des SEKA-Assemblers.

Besser Damit Sie von Anfang an gleich richtig loslegen können. Richtiger Einsatz von Maschinensprache an vielen praktischen Beispielen.

Schneller Durch den schnellen Erfolg können Sie schon nach kürzester Zeit Ihre eigenen Intuition-Fenster und die verschiedensten Diskettenoperationen problemlos programmieren.

Praktischer Auch wenn Sie nicht mit dem SEKA-Assembler arbeiten, zeigen wir Ihnen, wie man den SEKA-Source an andere Assembler anpaßt.

